CLEARWATER: AN EXTENSIBLE, PLIABLE, AND CUSTOMIZABLE APPROACH TO CODE GENERATION

A Dissertation
Presented to
The Academic Faculty

By

Galen Steen Swint

In Partial Fulfillment
Of the Requirements for the Degree
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
August 2006

CLEARWATER: AN EXTENSIBLE, PLIABLE, AND CUSTOMIZABLE APPROACH TO CODE GENERATION

Approved by:

Dr. Calton Pu, Advisor
College of Computing
Georgia Institute of Technology

Dr. Ling Liu
College of Computing
Georgia Institute of Technology

Dr. Karsten Schwan
College of Computing
Georgia Institute of Technology

Dr. Olin Shivers
College of Computing
Georgia Institute of Technology

Dr. Donald F. Ferguson
IBM Software Group
IBM Corporation

Date Approved: June 30, 2006

ACKNOWLEDGEMENTS

One of the most wonderful, extraordinary qualities of a computer program is its synthesis of ideas from many sources into a useful, operational whole. To a certain degree, I think of a computer program as a bit like a knowledge machine in which every programmer and every designer create the patterns and parts that comprise the whole. Perhaps not too surprisingly, just as computer programmers often must sledge, hone, and burnish our ideas into the precision-fit final design, so too do many take part in the creation of a neo-Ph.D.

Of my family, I wish to thank both my wife and my parents. My wife, Melissa, has had the fortitude, good nature, and stamina to keep me on track, to cheer me on when my motivation might flag. I thank her for that and hope I can re-pay her patience. My parents, too, were invaluable in their support, but more importantly, they set me on the path to achieve this Ph.D. – a concerted 29-year effort. I thank them for their guidance and motivation to move down the path to the highest education.

Calton, my advisor, is also key to my success as a student. Without his vision and perception, I might very well have labored in these gold mines without producing an ounce of refined product; there were numerous times when his perspective was vital to extracting the true value of my research for presentation. Most importantly, Calton never failed to provide the resources and opportunities for me to succeed.

Other students students and researchers have been instrumental in my Ph.D. work, as fellow programmers, idea generators, and social support. Thanks especially to Young, Wenchang, and Gueyoung for their tireless programming and willing technical discussions that led to the ISG and ACCT/Mulini. Thanks also to the DiSL group and

Enterprise Research Group for serving as test audiences for the presentation of material before conferences. Thank you also to Ling Liu, for her additional guidance and input and to Charles Consel for the opportunity to collaborate with him and study abroad.

Finally, in recognition of the funding without which this work could not have occurred: this work was partially funded by DARPA/IXO as a project in the PCES program, by DoE as a project in the SciDAC's Scientific Data Management Center, by NSF/CISE as a project in the CCR division's Distributed Systems program, IIS division's Data and Application Security program, and the ITR program. This work was partially supported by NSF/CISE IIS and CNS divisions through grants IDM-0242397 and ITR-0219902, DARPA ITO and IXO through contract F33615-00-C-3049 and N66001-00-2-8901, and Hewlett-Packard.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Since the advent of RPC Stub Generator, software tools that translate a high level specification into executable programs have been instrumental in facilitating the development of distributed software systems. Developers write programs at a high level abstraction with high readability and reduced initial development cost. However, existing approaches to building code generation tools for such systems have difficulties evolving these tools to meet challenges of new standards, new platforms and languages, or changing product scopes, resulting in generator tools with limited lifespan.

The difficulties in evolving generator tools can be characterized as a combination of three challenges that appear inherently difficult to solve simultaneously: the abstraction mapping challenge translating a high-level abstraction into a low-level implementation), the interoperable heterogeneity challenge stemming from multiple input and output formats, and the flexible customization challenge to extend base functionality for evolution or new applications. The Clearwater approach to code generation uses XML-based technologies and software tools to resolve these three challenges with three important code generation features: specification extensibility, whereby an existing specification format can accommodate extensions or variations at low cost; generator pliability, which allows the generator to operator on an extensible specification and/or support multiple and new platforms; and flexible customization, which allows an application developer to make controlled changes to the output of a code generator to support application-specific goals.

The presentation will outline the Clearwater approach and apply it to meet the above three challenges in two domain areas. The first area is information flow

applications (e.g., multimedia streaming and event processing), a horizontal domain in which the ISG code generator creates QoS-customized communication code using the Infopipe abstraction and specification language. The second area is enterprise application staging (e.g., complex N-tier distributed applications), a vertical domain in which the Mulini code generator creates multiple types of source code supporting automatic staging of distributed heterogeneous applications in a data center environment. The success of applying Clearwater to these domains shows the effectiveness of our approach.

CHAPTER 1
INTRODUCTION

1.1. Code Generation for Heterogeneous Distributed Systems

Code generation, including its use in RPC, has proven to be a useful tool for addressing challenges in distributed systems [10]. Developers can manipulate a high level abstraction which offers gains in readability and reduced initial development cost, and because of automatic translation, the low-level general purpose language implementation is also more likely correct than a handcrafted, one-off solution – developers are afforded empirically and sometimes formally "proven" code. Since then, domain specific languages and their generators have addressed myriad distributed computing problems such as inter-object communications [27], quality of service [69], application deployment [19][99], service-level codification [32][33][94], and "safe" in-network computation [49].

Unfortunately, the evolutionary pressure of internal and external considerations may limit a code generator's practical lifespan; evolving a generator's input language, incorporating new output targets, or customizing its output poses a significant challenge. Technology advancement necessitates change because new and unsupported APIs, software, or hardware replaces legacy systems. If standards are involved, adaptation cost may delay standards implementation until after the drawn-out standard process is complete. In doing so, developers trade timeliness for mitigated exposure to eleventh-hour "tweaks" that disruptively cascade through their design. Even within the context of a specific software effort, such as that undertaken by a research or application development team, inevitable refinements of goals and functionality translate into constant and costly generator refinement.

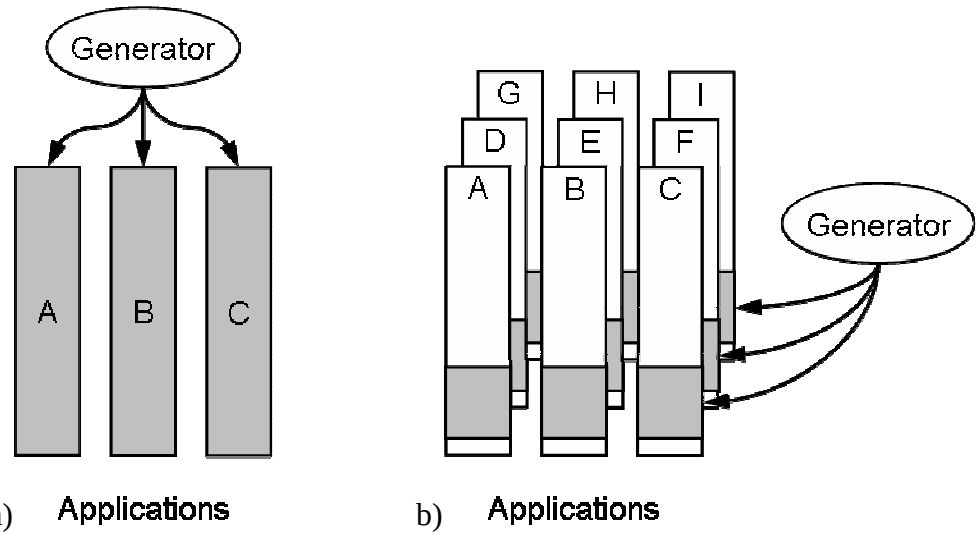


Figure 1. Vertical (applications) and horizontal (application layers) targets for code generation.

Heterogeneous distributed systems, in particular, demonstrate the pressures to which generators are subjected. Two of our recent research projects in distributed heterogeneous systems have required source-to-source translation from an evolvable specification, multiple language outputs, and customizable output. The first, the Infosphere project, targets the horizontal domain for specifying the communication structure between application units and translate this into source code with the Infopipes Stub Generator (ISG) [83][84][85][102][103][104][106]. The second, the Elba project, addresses a vertical domain in which the problems of automating enterprise application management and staging (pre-production testing and design verification) is addressed using two generators, ACCT and Mulini, to create suitable code artifacts from high-level policy documents [105]. In this case, a vertical domain refers to one in which a complete application is generated; a horizontal domain is one in which an application layer is generated as illustrated in Figure 1 [30]. While these generators contain a great deal of

domain functionality, at their current stage of development they primarily support source-to-source translation.

When generating Infopipes connections, there is a strong demand for both customized and multi-platform output from the ISG. For instance, two different Infopipes may both communicate through a socket but be implemented in different languages, say C and C++. Furthermore, these may both need customizations with quality-of-service code that can monitor latency or throughput and adjust the application behavior. Note that the customization is simultaneous across different programming approaches of C and C++.

Again with Mulini and ACCT, the following requirements appear: support input from multiple sources, allow for customized performance instrumentation in application source (Java), support generation to multiple platforms (scripts, build files, Java source, etc.). Mulini accepts as input a domain specific language describing the staging process and policy level documents, such as SLAs. From these, it generates several documents: scripts for automated deployment, performance monitoring code, analysis code, etc. When needed, application code can be promoted into the generator to be instrumented for application specific performance metrics.

Crafting a domain specific program generator capable of responding to these internal and external forces engenders overcoming non-trivial challenges in each stage of code generation.

1.2. Target Domains

The Clearwater approach was developed in the course of building the ISG for the Infosphere project. The second application domain, distributed enterprise application

management, is one of the first applications of the Clearwater approach to building a generator from scratch. In both domains, the code generator must create code supporting operations in a distributed, heterogeneous computing environment.

1.2.1. Information Flow Applications

The first generator, ISG, supports the Infosphere project; the chief concern was encapsulating middleware for distributed information flow systems, which are characterized by continuous volumes of information traversing a directed workflow network [86][59]. This pattern of communication and processing characterizes distributed systems such as high-volume e-commerce applications, distributed online games, digital media applications, and scientific and business monitoring systems. The Infosphere project was organized to develop tools, techniques, and methodologies for abstracting, building, managing, and reasoning about information flow applications and their demand for "live" information.

Information flows are streams of explicitly defined and typed data, and the data in the flow must be processed as it progresses from initial producer to ultimate consumer. Such processing may include transformation, storage, deletion, or computation of some metric about the stream itself.

Infopipes are the core abstraction used to encapsulate processing and communication for information flows. Infopipes may be simple, or complex, in which case they are compositions of simple pipes. A simple Infopipe instance has two ends – a consumer (inport) end and a producer (outport) end – and implements a unidirectional information flow from a single producer to a single consumer. Between the two ends is the developer-provided Infopipe middle. The middle encapsulates an application's

computational task for the data flowing through the Infopipe. From these simple Infopipes, more complex Infopipes may be constructed as serial or parallel compositions of simple Infopipes connected via their inports and outports. These complex Infopipes can then be used in the same manner as simple Infopipes.

As an abstraction, Infopipes are language and system independent; consequently, generated stub of code in the abstraction is able to hide the details of marshalling and unmarshalling parameters for heterogeneous languages, hardware, communication middleware, etc. Heterogeneity remains a concern even if there is a common binary layer, such as Java, as the platforms may differ significantly enough in hardware (say memory or power availability) that it has implications for application behavior. Therefore, the generator for Infopipes code must provide for customization by the application developer beyond a simple platform choice; this customization functionality must be compatible with the abstraction mapping and heterogeneity interoperability the generator provides.

Currently, there are three different tools for constructing Infopipes applications. The first is a GUI tool based on the Ptolemy II workflow editor. This tool emits an XML document that captures a general workflow graph. Second, there is a text-based language Spi (for Specifying Infopipes), and lastly an XML format, XIP (XML for Infopipes). Both the GUI representation and the Spi representation are converted to XIP prior to code generation. The GUI representation, producing an XML format, is converted via XSLT scripts, while Spi is converted a traditional parser that builds a parse tree in-memory and produces a straightforward XML representation of that tree.

1.2.2. Distributed Enterprise Application Management

The second domain is distributed enterprise application management. Application complexity in the enterprise is driving the creation of new tools that support the creation of verifiable and/or self-managing systems. In addition, new paradigms for assembling enterprise software, such as the service-oriented architecture, are accompanied by a proliferation of specifications that must also be implemented, tested, and deployed.

The first effort in the enterprise application management space addressed the resource deployment problem in which distributed applications should start efficiently and in provably correct fashion while enforcing serialization constraints and leveraging the distributed systems' inherent parallelism. The generator, ACCT, maps high-level, formal design descriptions into formats suitable for deployment engines, such as SmartFrog [45]. In this domain, ACCT helps "close the loop" in a feedback-based business-objective-driven management system for utility computing environments by bridging between the design and deployment of an application [94]. This pushes application deployment from the realm of brittle, uncertain, *ad hoc* scripts to provably correct and efficient automation.

ACCT accepts declarative, high-level input documents computed from constraint specifications; no previously available deployment tools operate directly from them. The high-level inputs for ACCT are created by Cauldron, a high-level reasoning engine [90]. MOF, the OMG Meta-Object Format, is used to describe applications, hardware, and encode their constraints. From a MOF document, Cauldron produces a new MOF document that provides a mapping of software onto hardware and also a pairwise dependency list for deployment and application startup. ACCT maps Cauldron's output into input for SmartFrog which can execute the deployment workflows.

Following the ACCT generator, the Mulini code generator uses policy level documents to create automated staging plans for enterprise applications. The complexity growth of enterprise applications naturally translates to testing those same applications. Furthermore, enterprise applications, especially those built around web services, also must implement multiple non-functional specifications such as service-level agreements.

Mulini accepts as input policy level documents such as service level agreements, constrained deployment plans, and staging description documents and creates scripts and instrumented source code which can be used in an iterative fashion to verify non-functional aspects of an enterprise application. Mulini also reuses the ACCT generator within it to support the creation of deployments for the application and staging-time tools specifically tailored to the staging environment.

ACCT and Mulini shares similar goals to the ISG: 1) translate high-level specifications to executable code; 2) support translations to multiple domains and supporting multiple enterprise tools, and 3) support formal verification of deployment schemes. Given the early stages of the Elba project, efforts have concentrated so far on the first two goals.

1.3. Code Generation Challenges

The two domains introduce a common set of problems in the implementation of these generators: (1) the heterogeneity of languages, operating systems, and hardware, (2) the translation between the high level abstractions to (many) low-level implementation layers, and (3) customization to particular instances, *i.e.* to a particular application or to a particular deployment environment.

Specifically, viable solutions for software tools for use in heterogeneous, distributed systems must address three significant challenges simultaneously:

- First, very high-abstraction descriptions must be mapped onto low-level platforms automatically. This means generators translate from high-level design tools such as a GUI into a general purpose language and communication layer appropriate to each target participating in the system. This is the *abstraction mapping challenge*.
- Second, because of heterogeneity in the system, a practical solution must accommodate inputs from multiple specification regimes and outputs to multiple target platforms. This is the *interoperable heterogeneity challenge*.
- Finally, the third challenge is that of providing a mechanism whereby each application developer can augment functionality created by the tools and introduce his own application-specific properties irrespective of the target platform. This is the *flexible customization challenge*.

Unfortunately, when considered in pairs the challenges become much more difficult due apparently inherent tradeoffs.

A high level of abstraction may successfully hide multiple target platforms from an information flow application developer, but the abstraction level becomes problematic if the mapping problem is attacked with traditional code generation techniques. Such a generator imposes a high cost because multiple layers must be maintained: a lexer, parser, intermediate representation, and also custom generators for each target platform. The toll for maintenance becomes especially apparent if the specification changes under

the influence of either external forces (*e.g.*, an updated standard) or internal forces (such as new research).

Too, the heterogeneity challenge to accommodate multiple input and output targets stymies customization. It is easy to see that limiting developers to only Java or only Windows .NET might motivate libraries of code that are mutable and customizable through sub-classing or byte-code morphing or even aspect weaving, but when requirements dictate interoperability between Java and .NET, or the application includes a mix of older C or C++ code, then platform or language specific options are unsuitable. Obviously, too, falling back to manual techniques for customization is undesirable: generated code may be abstruse, especially if optimized in some fashion; manually written code is error prone; since it is easily obscured by generated code it is more difficult to maintain; and manual customizations can be lost if an abstraction change triggers re-generation of the application.

Finally, a high abstraction level and the need for customization engender an inherent trade-off: customization is the accomplishment of detailed, application- and platform-specific changes to code, but a high abstraction deliberately hides such details. Writing great volumes of generic code to address all possible application cases on all possible platforms and with all possible parameters is a programming quagmire that would further demand an abstraction language burdened by complexities, quirks, and details irrelevant to many applications.

Obviously, given the problems between the pairs of challenges, resolving them together, with one tool or tool suite, is much more difficult.

A simple Infopipes application readily illustrates the tradeoffs. Imagine that a computationally powerful video source is sending to a device, like a cell phone, that is power and CPU constrained. It is desirable to slow the sending rate of the source to avoid overrunning the phone's capabilities. The relation between the two is a simple Infopipes composition of one Infopipe with and outport (the serving side) and one Infopipe with an input (the consumer). Each "half" of the application demands different customizations: on the server side it must be customized with rate controlling code and on the receiver side with resource monitoring code tailored to that particular receiver. The challenge is to create a generator that supports all of the previous within a single framework.

The challenges, however, also point towards using code generation as a solution. Code generation offers the possibility of language independence and therefore abstracts over some heterogeneity. Of course, code generators by definition provide abstraction mapping from a specification domain into an implementation domain. To overcome the three challenges, there are three identifiable requirements.

1.4. Solution Requirements

Meeting demands of code generation in the ISG, ACCT, and Mulini requires a high-level specification, its translator, and its output to support three features:

Specification extensibility — Extensibility is the ability for domain experts to add to the high-level domain abstraction, *i.e.*, to extend the domain language, with minimal impact on pre-existing specifications. Furthermore, the generator should support a variety of domain-level input sources with a common tool (text files, program toolkits, GUIs, etc.).

Generator pliability — Pliable generators are also flexible – they can support multiple input and output formats [37] and also the extensible language requirement above. This means the generators should be robust to changes in input specification, *i.e.* specification changes should require no or minimal re-writes to the generator. An effect of writing such a code generators is that generator implementation need not be complete. For example, the implementation may only understand a portion of the input specification. This aids in the writing of generators that stretch the generators functionality to new target platforms.

Flexible customization — Flexible customization affords the application programmer opportunities to make changes to generated code to match his particular application requirements in an aspect oriented fashion. For instance, quality of service often demands such consideration. Too, a staging administrator may wish to customize output to support application instrumentation. Such changes may be application-specific and therefore not suitable for general inclusion in the code generator. Supporting modularity encourages the writing of reusable modifications for the generated code.

Flexible customization warrants further discussion as particularly in heterogeneous distributed systems, the resultant code from these generators often needs customization. For example, differing `signal()` conventions hamper Unix application portability. Promoting such relatively small variations into the generator itself might

needlessly complicate generator development and maintenance since the customization must either be implemented across multiple target platforms. On the other hand, not supporting the needed feature at all implies a developer customizes the output code manually. Manual customization sacrifices reuse of the custom code and the changes may be easily lost, but adapting the generator may demand a disproportionately large effort for an otherwise minor enhancement.

The result of these efforts, the Clearwater approach, uses XML [14] and XSLT [23] to provide customization and allow for evolution in the input language while accommodating differing target platforms. It has three major features: specification extensibility, generator pliability, and output modularity.

So far, Clearwater has guided construction of three non-trivial code generators for use in two different domains. The first domain is the Infopipes Stub Generator, or ISG, an application layer generator that generates and weaves customized communication code for information flow applications. Code generators in this domain benefit customizable generated code and extensible domain specifications

The second domain is enterprise application management. Within that domain, the ACCT generator, a deployment automation tool for built-to-order enterprise applications that maps verified designs into heterogeneous languages needed by deployment workflow tools. The Mulini generator supports the Elba project with a goal of automating the staging process.

Experiences with many differing output formats in the both the ISG and ACCT suggest that the Clearwater approach generally is not limited to any particular input or output language. The ISG underpins four types of input: Spi, a human readable format for

Infopipes; Ptolemy II, a GUI builder for workflows; XIP, the XML description of Infopipes and native format for ISG; and WSLA, the Web Service Level Agreement specification. ACCT, which is less mature, supports CIM-MOF. For output, the ISG generates C, C++, and Makefiles [103], and ACCT generates Java and SmartFrog's specification language [91].

1.5. Thesis

The thesis of this dissertation is that using XML, XSLT, and XPath for code generation supports the building of code generators that meet challenges inherent in solving some problems found in distributed heterogeneous domains. In particular, traditional approaches to source-to-source translation face the challenges of abstraction mapping, interoperable heterogeneity, and flexible customization. These three challenges are solved in the Clearwater approach which is defined by its support for extensible domain specifications, pliable code generation, and flexible customization. Note that this dissertation is confined to issues related to code generation in particular rather than more general problems regarding domain specific language processing. These contributions are outlined in detail in subsequent chapters.

In the next chapter, these code generation challenges are explained as well as why the challenges are difficult to resolve within a single solution. Then, the Clearwater approach is presented; the presentation includes an architectural overview of a Clearwater code generator, the XML-based software tools used in implementing Clearwater generators, and the three important Clearwater features mentioned – extensible specifications, pliable generation, and flexible customization.

Following the presentation of Clearwater, chapters three and four present the implementations of Clearwater generators for two domains and evaluations of the generators in exemplar applications. The first domain and its generator, the Infopipes Stub Generator, creates communication software for information flow applications. The output is customizable for supporting QoS and other application-specific requirements via the AXpect weaver, and benchmarks show the produced code to be performance-comparable with other communications software packages. The second domain, Distributed Enterprise Application Management, is supported through two generators, ACCT and Mulini. These generators also produce customizable output to support tailored code for application deployment (ACCT) and application staging (Mulini).

Chapter five examines the efficacy of the Clearwater approach towards increasing code reuse. While there are only a few Clearwater generators to use for data, the experience embodied by the Infopipes Stub Generator, ACCT, and Mulini indicate the Clearwater generators can achieve significant code reuse. In particular, in the Infopipes Stub Generator, extending the generator to new output targets seems to be of reasonable cost; in Mulini, Clearwater enabled the wholesale extension and reuse of the ACCT generator within the Mulini generator.

Finally, this dissertation will conclude with a summary and discussion of the implications of Clearwater research. There are several interesting questions remaining regarding the application of Clearwater principles to domain specific languages and their processors – for instance, can a processor provide verifiable functionality and still support extensible specifications? Other interesting issues involve leveraging the Clearwater architecture further. For instance, it seems likely that XSLT can facilitate

system-level weaving of specifications for pre-generation customization. This would allow more sophisticated handling of QoS, for instance.

CHAPTER 2
CLEARWATER

This chapter first introduces and discusses a Clearwater generator's relation to traditional compiler architecture; following this is a presentation and discussion of how XML and XSLT provide specification extensibility, generator pliability, and output modularity inside that model. While the architecture may mirror those found in traditional code generators, traditional implementation techniques rely on developing a language and grammar, parsing inputs into a token stream, building a custom abstract syntax tree (AST), and then tailoring a code generator to the AST.

While this monolithic implementation strategy has some benefits in terms of simplicity as all pieces are built at once, it also leads to code generator designs in which parts of the code generator easily tightly connected. Consequently, a change to or extension of the specification language requires multiple simultaneous activities: creating the new domain language features, defining their lexical patterns, defining their grammar rules, updating the AST design, and finally, reconciling the generator to the new AST. Only when the developer has completed all these can he construct a demonstration application and test the new produced code – a non-trivial task on its own. If multiple targets are required, the developer must change and test the generator for each and every target (implementation) platform.

This overhead proscribes specification extensibility since it magnifies even small changes; generator pliability is limited since a language change must propagate through multiple platforms. Code modularity is not readily addressed in any platform independent

fashion, either. On the other hand, Clearwater generators, based on the use of XML and XSLT, do have these capabilities.

2.1. Clearwater Features

2.1.1. Architectural Overview

From an architectural viewpoint, a Clearwater generator has multiple serial transformation stages – it is a code generation pipeline. The Clearwater hallmarks are that stages typically operate on an XML document that is the intermediate representation, and XSLT performs code generation. The overall process:

1. Compile developer-centric format to an XML-based intermediate format (High Level Language-to-XML), mainly a straightforward translation. In terms of a text-based format such as Spi, it can be accomplished by building a parse tree and converting it directly to an XML representation. In terms of a GUI tool, XSLT is used to convert the Ptolemy II representation into a XIP document.
2. Pre-process the XML intermediate representation. This involves looking up extra information from disk, if needed, resolving names, *etc.*, and adding the new information into the XML intermediate representation. In doing so, this maintains the intermediate representation as an XML document.
3. Generate code via XSLT that transforms and augments the XML intermediate representation with source code yielding an XML+Source code specification. In this step, the XSLT templates also insert additional XML tags along with the source code to be used in the next step. One might also consider this as a parse tree annotated with source code.

4. Weave the source and specification with any aspects. This step may involve iterative code generation and weaving steps that consume and produce XML elements containing output source code.
5. Write generated source to files and directories transforming XML documents containing source code into pure source code.

In a Clearwater generator, stage two reads and parses an XML input file to produce a DOM (Document Object Model [62]) tree in memory, a decoupling that facilitates one generator's serving multiple high-level languages since they need only compile to an XML format. In practice, Clearwater-based implementations have kept the high-level compilers of stage one independent from steps 2 through 5 and use the XML intermediate format as the primary input for experimentation as this allows for greater flexibility in terms of research. However, an implementation might easily wrap step 1 and steps 2 - 5 in a shell script. Stage two also preps the intermediate language for processing by the code generator

2.1.2. XML: Extensible Domain Specification

XML's chief contribution to the Clearwater approach is that it introduces extensibility at the domain-language/domain-specification level. This stems from XML's simple, well-defined syntax requirements and ability to accept arbitrary new tags thereby bypassing the overhead encountered when managing both a grammar and code generator.

Specification 1

```
<datatype name="FloatArray">
  <arg name="SIZE" type="integer"/>
  <arg name="buff" type="string"/>
</datatype>

<pipe name="UAV">
  <subpipes>
    <subpipe name="Sender" pipeOf="Sender"/>
    <subpipe name="Receiver" pipeOf="Receiver"/>
  </subpipes>
  <connections>
    <connection comm="ECho">
      <from pipe="Sender" port="out1"/>
      <to pipe="Receiver" port="in1"/>
    </connection>
  </connections>
</pipe>
```

Specification 2 - Extended

```
<datatype name="FloatArray">
  <arg name="SIZE" type="integer"/>
  <arg name="buff" type="string"/>
</datatype>

<filter name="GREY">
  <in type="ByteArray"/>
  <out type="ByteArray"/>
</filter>

<pipe name="UAV">
   <subpipes>
      <subpipe name="Sender" pipeOf="Sender"/>
      <subpipe name="Receiver" pipeOf="Receiver"/>
   </subpipes>
   <connections>
     <connection comm="ECho">
       <from pipe="Sender" port="out1"/>
       <to pipe="Receiver" port="in1"/>
       <use-filters>
         <use-filter name="GREY"/>
       </use-filters>
     </connection>
   </connections>
</pipe>
```

Figure 2. Specification 1 is a fragment from a basic Infopipe specification extended, without modifying any grammars and using the same parser, to include the ‘`filter`’ construct and ‘`use-filter`’ modifier as in Specification 2.

As an example of specification extension, consider a scenario in which a developer adds new information specific to a target architecture. In Infopipes, an example is that native sockets support only data transmission, but the ECho event middleware supports “safe”, uploadable filters on events [38][39]. To accommodate the filter functionality at the domain level, the ECho developer must first extend the specification with new filter descriptions, as illustrated in Figure 2. Whereas the use of a grammar based approach encounters the difficulties listed in the introduction, in the Clearwater approach adding new elements to the specification document alongside existing elements requires no changes to the parser, lexer, syntax checker, or grammar definition.

In maintaining grammars, a developer spends a great deal of time explaining the domain language structure to the parser by defining tokens (lexing) and their valid orderings. Deviations from defined rules break the lexer/parser; experimentation and specification evolution become difficult. Furthermore, most generation approaches create an abstract syntax tree based explicitly on the grammar for the language. Therefore, any language change finds its way into the AST, and from there the code generation logic that interacts with the AST must *also* be changed.

Because XML always represents a fully-parenthesized syntax tree, document structure is always explicit (through element nesting and angle brackets), and rules that govern the structure are (often) implicit. Consequently, a changed specification format very often can be accepted without syntactic complaints by the existing generator package. This extensibility sidesteps the problems of parsing by isolating them from the code-generator chain. Because XML documents implicitly encode production rules, developers of domain language generators benefit by avoiding the premature tying of the

generator to a particular concrete grammar. Users can add new XML tags to a well-formed XML document, and therefore to their language grammar, provided the changes maintain well-formedness.

While an XML document is an enforced hierarchy of tags, the content of the document is not limited to expressing hierarchical relationships. Elements can refer to other portions of the document through names and identifiers. For instance, a single simple Infopipe's definition is reused multiple times by using its name to look up data as needed from a shared definition.

XML has several advantageous properties for being a general specification format. First, XML defines a very simple lexical pattern for characters that allows automatic tokenization by the XML document parser. Reserved words which create a "block" of code with some meaning are either (1) enclosed in angle brackets and given the meta-name "element" (*e.g.*, '`<subpipes>`' in Figure 2), or (2) form a quote-delimited name-value pair specific to an element and forms an "attribute" (*e.g.*, '`name="UAV"`'). New reserved words can be added to a language by adding new elements or attributes to the XML representation. XML itself only reserves two symbols, '<' and '&', the first to identify elements and the second as an escape character.

Extensibility's great advantage during ISG development lay in its supporting multiple researchers' efforts simultaneously with minimal concern for specification mismatches. As it turned out, each team researcher created a slightly different code generator that operated from the same core XML document. For instance, one developer worked on support for aspects (AXpect) and introduced tags to support that effort while another developer worked on mobile data filters with his own custom tags added to the

core document. Importantly, the developers could reuse the documents of each other for various testing purposes without worrying about breaking their own code.

To facilitate reuse of Infopipes specifications, the ISG stores declarations for later reuse at which point they may be invoked by name. This persistent data is stored as files on the system. XML simplifies the process of storing since each declaration block can be stored as its own XML document and re-loaded from disk without invoking a domain-language specific parser (only the XML parser).

Concluding the XML discussion, one last useful feature, though not strictly germane to fulfilling extensibility, is the XML namespace. An XML namespace, in principle, performs for XML elements the same function as a namespace in a general language, partitioning meaningful tokens into non-colliding groups. In practice, this means that several overlapping trees of information can exist in the same document. Each type of information, for instance information pertaining to quality of service, can be placed in its own namespace. If hypothetically, one were to include QoS information with an Infopipes specification, then the "`qos:connection`" element, which may hold quality of service information for a particular Infopipe communication link, remains in a namespace assigned to "`qos:`" separate from the XIP "`connections`" element which remains in the document's default namespace. (This is a very simplified presentation of namespaces. Readers are referred to the XML namespace standard for a full discussion [13].)

2.1.3. XSLT: Pliable Code Generation

In addition to the extensible specification, the use of an extensible specification demanded a pliable code generator. *Pliability*, is the ability for a code generator to

tolerate changes to the AST and readily support to serve new target platforms. Easily supporting new targets turns out to be a natural consequence of a pliable generator because new target outputs or functionality can be added to the generator on an as-needed basis and because each target need not support or be aware of the full domain abstraction. The Clearwater approach fulfills both of these requirements by using XSLT and its co-standard XPath [24].

XSLT, the Extensible Stylesheets Language for Transformations, is a (Turing complete) language for converting XML documents into other types of documents – typically another XML or HTML document. Each XSLT script, or stylesheet, is a collection of templates, and in the Clearwater approach, each of these roughly corresponds to some unit of transformation from specification to generated code. Practically, the pliability requirement means that XSLT generator code must have the ability to ignore unknown tags and still generate correct code that implements a portion of the input specification. It is the use of XPath that infuses XSLT with its flexibility; XPath allows a developer to refer to locations and groups of locations in an XML tree similar (syntactically) to how a hierarchical file system allows path specification. It has several important features improving beyond basic file paths, however.

First, XPath has a '`//`' ("descendant-or-self") 'axis' that encourages writing structure-shy paths [63]. A structure-shy path is one that is not closely tied to the absolute ordering and nesting of nodes in a tree. The '`//`' and the structure-shy qualities of XPath allow a developer to perform references to information without regard to explicit placement. Second, instead of each XPath statement referring to a single, unique node, the statement refers to some set of nodes in the XML document. (Here, the use of the

more general term “node” instead of “element” is because XML attributes and text data are contained in nodes accessible through XPath.) Often, working with a set of nodes is desirable, as in the case of processing data type fields where one may loop through each member to generate a declaration. When sets of nodes are not desired, XPath’s predicate functionality, the third important XPath feature allows the generator developer to reduce a set to a single node.

Figure 3 illustrates that moving data-descriptions within the document does not break a properly written XPath statement that retrieves that data from a datatype declaration located in various places within the specification document. A language developer faces a choice of to which scope a datatype declaration should be bound. Global datatype declaration, in the first panel, affords the best possibilities for reuse later in the document. Infopipe-level declaration reduces the reuse possibilities, but also affords developers options for changing the datatype at generation time without affecting other dependent Infopipes. In the third panel, the datatype is bound to the scope of a communication link, which offers the possibility of per-link customization of the datatypes. In all three cases, the same XPath statement returns the information contained in the datatype declaration.

XPath: `//datatype[@name='ppmType']/arg[@type='long']`

```
<datatype name="ppmType">
  <arrayArg name="mag"
            type="char" size="2"/>
  <arg name="width" type="long"/>
  <arg name="height" type="long"/>
  <arg name="maxval" type="long"/>
  <arg name="pictureSize"
       type="integer"/>
  <arrayArg name="picture"
            type="byte"
            size="pictureSize"/>
</datatype>

<pipe lang="CPP"
      class="ReceivingPipe">
  <apply-aspect
       name="receiver_gpce.xsl"/>
  <ports>
    <inport name="in"
             type="ppmType"/>
  </ports>
</pipe>
```

```
<pipe lang="CPP"
      class="ReceivingPipe">
  <apply-aspect
       name="receiver_gpce.xsl"/>
  <ports>
    <inport name="in"
              type="ppmType">
      <datatype name="ppmType">
        <arrayArg name="mag"
           type="char" size="2"/>
        <arg name="width"
              type="long"/>
        <arg name="height"
              type="long"/>
        <arg name="maxval"
              type="long"/>
        <arg name="pictureSize"
            type="integer"/>
        <arrayArg name="picture"
            type="byte"
            size="pictureSize"/>
      </datatype>
    </inport>
  </ports>
</pipe>
```

```
<pipe lang="CPP"
       class="ReceivingPipe">
  <datatype name="ppmType">
    <arrayArg name="mag"
              type="char" size="2"/>
    <arg name="width" type="long"/>
    <arg name="height" type="long"/>
    <arg name="maxval" type="long"/>
    <arg name="pictureSize"
         type="integer"/>
    <arrayArg name="picture"
      type="byte" size="pictureSize"/>
  </datatype>
  <apply-aspect
          name="receiver_gpce.xsl"/>
  <ports>
    <inport name="in"
            type="ppmType"/>
  </ports>
</pipe>
```

Figure 3. Here, the XPath expression returns all the data members of type '`long`' for the type '`ppmType`' in all three cases even though datatype has been moved within the specification document.

In operation, a language developer can write a template to be activated in one of two fashions. First, the template may be invoked explicitly by name – this is just as one calls a procedure or function in other languages. Second, the template may be invoked implicitly by an XPath pattern match. Pattern matching consists of two parts: the selected nodes and template matches. When the developer reaches a point in the template where he intends to invoke further generator functionality, he writes an '`apply-templates`' statement that selects a nodeset. The nodeset is then treated as a list of nodes for possible processing. Then, the XSLT engine processes each node in the list by matching it to the template that has the 'best match', where best match is based on specificity and priority as defined by the XSLT standard. In the method of execution, a general pattern that selects nodes for processing may end up triggering many different templates in the generator.

Consider the ISG code generator's operation over a XIP document. The document can be represented as a tree with a root element '`xip`', containing sub-elements. The '`pipe`' sub-element encapsulates the data that describes an Infopipe. Then, to execute generation for pipes:

First, the developer writes the selection pattern to extract the pipe tags into a nodeset:

```
<xsl:apply-templates select=”/xip//pipe”/>
```

This statement is blind to the fact that each pipe is potentially going to need a different set of language and communication templates. Effectively, the platform specific information for this level of the generator has been abstracted over using XPath. If the

specification contains the following two elements, one indicating a C target and the other CPP, both will match the above pattern and placed in a list of nodes for processing:

```
<pipe name="imageSource" lang="C">…
<pipe name="imageReceiver" lang="CPP">…
```

The XSLT processor then consults the list of templates for possible matches. If there are two templates, say one for C, one for C++, and one for Java, they will be present in the XSLT as:

```
<xsl:template match="/xip//pipe[lang='java']"/>…
<xsl:template match="/xip//pipe[lang='C']"/>…
<xsl:template match="/xip//pipe[lang='CPP']"/>…
```

Once matched, the C template is invoked once on the C Infopipe specification, the C++ template once on its corresponding specification, and the Java template is not invoked at all.

From this, one can see how pliable support for multiple targets naturally emerges when runtime compilation, pattern matching, and stylesheet importation combine. In the ISG, language-specific XSLT files are imported into a single `masterTemplate.xsl` file, and pattern selection from the specification controls the execution. The same approach applied to the communication layer level to support differing communications packages organizes generator code into manageably sized templates and files.

The first enabler for multiple platforms is XSLT's provision for invocation by either call-by-name and pattern matching. The makes it possible to alternate control of the generation process between the generator and the specification. For example, using a pattern to match the C Infopipes, as above, lets the specification control entry into that

group of templates. These templates may call by name other templates that automatically generate header files and make files – at which time the generator-code controls the code production. With the ISG, it is common to use both. Frequently, call-by-name templates are used to separate code generation into smaller fragments when a single pattern match may trigger lot of code is to execute.

Second, XSLT also supports importation of stylesheets, as shown in Figure 4, so that complex stylesheet behavior can be composed from multiple, simpler stylesheets. Alternatively, a complex stylesheet can be broken into smaller stylesheets for better organization and refactoring of generator code. As an example of this technique, the ISG keeps separate stylesheets for C and C++ generation and further deconstruct those into smaller stylesheets based on the communication mechanism supported (e.g. TCP or the ECho middleware package).

```
masterTemplate.xsl
<xsl:import href="allMake.xsl"/>
<xsl:import href="CPP/CPP.xsl"/>
<xsl:import href="C/C.xsl"/>
...
<xsl:apply-templates select="/xip//pipe"/>

C.xsl
...
<xsl:template match="/xip//pipe[@lang='C']">
...

CPP.xsl
...
<xsl:template match="/xip//pipe[@lang='CPP']">
...
```

Figure 4. By inserting an import directive and using XPath pattern selection for the target language, extension to new output targets is easy and independent.

Finally, XSLT is runtime compiled allowing output to change easily and quickly. One might mimic this functionality through external resource strings if developing in a compiled, object-oriented environment like Java in a technique similar to what is done for internationalizing applications interfaces. That is, the generator developer might load strings on demand from disk which are then translated into the output, but generator development then becomes limited to variations on pre-identified output strings. Consequently, any reorganization that does not already fit the established mapping from high-level language to the implementation language will require changes to a generator object. If the generator then had original strings for generating just functional target code, then the move to an object-oriented language would be unable to support the natural OO paradigm of classes and inheritance.

Runtime compilation allows easy change of the output without re-writing objects or re-compiling. Generally, from experience, it is reasonable to make debugging changes from the output application directly to the generator templates and then re-generate the entire application. This shortens the development cycle and also lowers the maintenance hurdle. While the generator may not be quite as fast as a compiled generator, for the programs generated so far, it compares favorably to the application's compilation time, and therefore the speed of code generation is not a substantial impediment to application development.

Figure 5 on the following page provides a substantial template excerpt for generating Infopipes startup C code.

```
int <xsl:value-of select="$thisPipeName"/>( ) {
  <jpt:pipe point="user-declare">
  ; // USER DECLARES VARS HERE
  </jpt:pipe>
  <jpt:pipe point="user-function">
  ; // USER CODE GOES HERE
  </jpt:pipe>
  return 0;
}
// startup all our connections
int infopipe_<xsl:value-of
               select="$thisPipeName"/>_startup()
{
  <jpt:pipe point="startup">
  // start up outgoing ports
  // <xsl:for-each select="./ports/outport">
  infopipe_<xsl:value-of
               select="@name"/>_startup();
      </xsl:for-each>

  // start up incoming ports
  // <xsl:for-each select="./ports/inport">
  infopipe_<xsl:value-of
      select="@name"/>_startup(); </xsl:for-each>
  </jpt:pipe>

  return 0;
}
```

Figure 5. Excerpt from a template that generates connection startup calls and skeleton for the pipe's function. Line breaks inside XSLT tags do not get copied to the output.

2.1.4. XML+XSLT: Flexible Customization through Weaving

The final goal, modular code generation, is reached by combining the strengths of XML and XSLT to support aspect oriented XML weaving, and through that support flexible customization of the generated code. Since every XSLT document is valid XML, using the Clearwater approach allows one to embed new XML tags in code-generating XSLT without affecting the transformation process. Then, when this XSLT template generates output code, these XML tags are replicated into the target code where they act as semantic markers to expose the domain structure of the generated code. In effect, these semantic markers allow an application developer to be able to see through the generated code to the hidden domain structures it expresses. Each marked block of generated code becomes a module customizable through replacement or augmentation. In the ISG, the weaving capability is implemented by the AXpect weaver.

The aspect abstraction provides good encapsulation for the customizations to generator code [57]; it is frequently used to perform customization to achieve some goal orthogonal to generator's primary domain. In an information flow application, quality of service tends to crosscut multiple Infopipes and multiple inports and outports. Similarly, "security" is not a feature easily captured in XIP which is primarily concerned with expressing the communication patterns between and compositions of Infopipes. Implementing security, furthermore, requires changes in multiple places within the Infopipe: startup (key initialization), shutdown (clearing memory), and data transmission (encryption/decryption).

Another example of orthogonal operation is the application-specific functionality in each Infopipe middle where code performs the transformation/computation/storage on

the data, which is a generally orthogonal problem to the issue of transmitting the data in the information flow. The XML weaving technology can be used to generate the entire application rather than insert the middles by hand after generation. Bringing the computation code in with the generator also allows for parameterization of the middle code from the specification, as for example when the middle must look up data from a location that may vary by system, and assists in the overall automation of the development process since it eliminates the need to alter the produced files by hand.

AOP systems are characterized by a weaver that enables orthogonal functionality to be encapsulated as *advice* which is executed when the weaver matches a *joinpoint* in a program, where the advice will be merged into the base code, to a *pointcut* in the aspect that determines when a given piece of advice should apply [57]. In general purpose language systems, *e.g.*, in AspectJ, pointcuts often target joinpoints tied to the execution of a program. Such advice, for example, may be executed when a log object's print method is called by a web servers socket object. In the case of domain specific languages and AOP, such as Bossa [7] or the AXpect weaver, joinpoints are related to tasks or features naturally present in the domain decomposition. For example, in Infopipes there is provided a joinpoint for marshalling parameters.

However, XML, XSLT, and XPath combine to make the mechanics of these code substitutions and additions easy for Clearwater weavers. Given a generated document with the aforementioned XML tags, an XSLT template can use XPath to find those tags and replace or augment the existing code with new code and tags. From an AOP vantage point, XPath selects pointcuts and XSLT encapsulates advice over the joinpoints. The XSLT processor performs the task of joinpoint identification and weaving. Note that the

only language dependency in this process is the direct dependency between the advice and the target source language so that language-specific weavers are bypassed. The AXpect weaver works equally well on C and C++ Infopipes.

Consider the excerpts in Figure 6. The `jpt:pipe` tags in the generator template denote the code that performs shutdown tasks for an Infopipe which consists of successively shutting down inports and outports. On the right, the tags are kept with the code after generation and clearly label the purpose of that block of C code. From an AOP perspective, these tags form a set of joinpoints on the underlying generated code. Each joinpoint maps some logical domain feature into the “physical” implementation in a target language.

Generator Template

```
// shutdown all our connections
int infopipe_<xsl:value-of select="$thisPipeName"/>_shutdown()
{
  <jpt:pipe point="shutdown">
  // shutdown incoming ports <xsl:for-each select="./ports/inport">
  infopipe_<xsl:value-of select="@name"/>_shutdown(); </xsl:for-each>
  // shutdown outgoing ports <xsl:for-each select="./ports/outport">
  infopipe_<xsl:value-of select="@name"/>_shutdown(); </xsl:for-each>
  </jpt:pipe>

  return 0;
}
```

Emitted XML+Code

```
// shutdown all our connections
int infopipe_sender_shutdown()
{
  <jpt:pipe point="shutdown">
  // shutdown incoming ports

  // shutdown outgoing ports
infopipe_ppmOut_shutdown();
  </jpt:pipe>
  return 0;
}
```

Figure 6. "Generator Template" displays the XML markup in the XSLT; "Emitted XML+Code" shows how the markup persists.

There are two major benefits from this. First, it allows code generation to be modular. If an application's requirements demand replacing default generated functionality, this mechanism makes it possible. For instance, Infopipes exchange connection information via NFS files, but emulated distributed environments require hard-coded connection information for network interface specificity.

Second, it allows a developer to insert features into the generated code that are otherwise orthogonal to the domain language. A good example of an orthogonal feature encapsulation is a WSLA governing Infopipe performance, as in the first application example for the Infopipes generator [102].

Joinpoints fall naturally into two broad categories – those that pertain only to the domain, and those that pertain only to the implementation targets. This dichotomy is beneficial because often an aspect may introduce structures or behavior dependent on syntactic structure rather than just implementing domain function. For instance, both the C and C++ versions of Infopipes have similar marshalling and unmarshalling code, but if a new variable is to be introduced for use in the marshalling and unmarshalling process, then in C, it is more natural to make the variable a static, file-scoped variable whereas in C++ the idiomatic approach is to make it a member of the correct class. This distinction will be elaborated further in later discussion of the AXpect weaver.

2.2. Implementations

There have been three major Clearwater generators developed thus far. First, for supporting Infopipes applications, the ISG, which also drove the development of the Clearwater approach; it converts Infopipe specifications, XIP, into general purpose language implementations and supports AOP via its AXpect module. The second domain,

distributed enterprise application management, supported by ACCT and Mulini, resulted from collaboration with HP Labs. Though both are newer and less developed, both are still built upon the XML+XSLT approach of Clearwater.

2.2.1. A Quick Look: The ISG

The current version of the ISG generator uses a C++ based XML parser and DOM document interface with an embedded XSLT processor. The need for a general purpose language stems from the discovery of two limitations of employing pure XSLT. First, output file support is limited, and while new standards enable multi-document output, this was not true at the time of early versions of the ISG. Second, XSLT has only recently added the capability of accessing computed XML document fragments at run-time. This limits the ability to construct XML fragments with information from a document in any sort of recursive fashion.

Instead, a C++ package and XML parser performs pre-generation processing, which involves resolving connections between Infopipes and retrieving specifications from the repository. This process specifically involves recursively descending through Infopipe descriptions and retrieving multiple documents from disk from the repository which were then melded together to form a XIP+ document, which contains the XIP declarations from inputs plus connection information, actually used for generation. For the same reason, the AXpect weaver is implemented through a C++ package that repeatedly calls the aspect XSLT templates.

One ISG goal was to support multiple communication layers and implementation languages simultaneously. In the ISG, C and C++ can be created concurrently from a single specification. For example, a C Infopipe may communicate via ECho event

channels to a second C Infopipe, which in turn sends data over a TCP connection to a C++ Infopipe.

2.2.2. A Quick Look: ACCT and Mulini

Instead of using a C++ harness like the ISG, both ACCT and Mulini use Java to manage generation, although the first version of ACCT was pure XSLT. ACCT itself is over 2000 lines of code incorporated in three major stages: 1) pre-processing to convert MOF to an XML format, XMOF 2) data extraction, and 3) translation to code for deployment via SmartFrog.

SmartFrog requires ACCT to produce three types of files. First, a workflow file is created by converting pairwise event dependencies emitted in Cauldron MOF into totally-ordered and properly synchronized events in the SmartFrog workflow language. The other two files SmartFrog needs are ACCT-generated component definitions written in Java. These are also from the MOF; they define generic component functionality and corresponding instances of components that represent the fully parameterized (needed at run-time) definitions of the components. These generated specifications are wrapped into a single XML format called XACCT (XML for ACCT) that should provide flexibility for other deployment tools in the future. Finally, one more XSLT template strips the XML and provides the ultimate conversion into SmartFrog-deployable sources.

Mulini, like ACCT, is built using Java. Again, the inputs are wrapped into a single specification document by Java methods. For Mulini, the input preparation involves the extraction of data from multiple policy-level documents based on directives in the master XTBL specification. The resulting XTBL+ document is then suitable input for the Mulini generators as well as the incorporated ACCT generator. Mulini output spans several types

of files as demanded by the staging task which involves the coordination of multiple types of tools and programs.

2.3. Work Related to Clearwater

The work most closely related to the architecture of the ISG is its architectural similarity to compilers. Also, it adopts an intermediate format for flexibility like `gcc` and Flick [37]. However, there are several important differentiating features. First, traditional compilers only map into basic assembly code. Flick, too, is restricted in its output abilities because it does not maintain a system state document as in the ISG with XIP+. This is crucial in achieving the flexibility to do code weaving. SourceWeave.NET is also similar in that it is a cross-platform weaver, but pertains only to .NET platforms [53].

The Polyglot project has focused on creating extensible high-level languages [77]. However, while Polyglot has seen use in other projects, users are limited to variants on Java syntax whereas the use of the Clearwater approach permits the use of any human-friendly syntax which can then be compiled to an XML intermediate format.

XML-based techniques and code generation have also been useful for extracting data from web pages. The XWRAP Elite wrapper-generator used a custom language similar to XSLT and a path specification syntax with some of the features of XPath to allow flexible extraction of data from web pages [65]. While Clearwater adds the significant feature of flexible customization, the XWRAP Elite system supported sophisticated analysis of the source documents and developer interaction for identifying information of interest. In effect, each source document was an extensible specification and the XWRAP Elite tools assisted the developer in extracting and discovering the information contained in the specification.

The use of XML and XSLT in Clearwater is not unique, but is not in apparent widespread use. XML+XSLT is advocated for code generation tasks in industry as well [95]. Karsai discusses a number of possible shortcomings in using XSLT+XML in a semantic translator [55], but Clearwater generators generally find the two technologies to be quite amenable as a basis for code generation and this has some support in the experience of some other reported research.

The SoftArch/MTE [48] and Argo/MTE [17] projects have also used XML + XSLT for code generation. Their project has primarily concerned with resolving mismatches between software engineering tools and uses XML + XSLT generators to "glue" off-the-shelf applications together. ISG results corroborate their experience. In addition, Clearwater generators explore issues beyond this including specification extensibility, generator pliability, and aspect weaving for modular output.

When compared to other industry tools, there are parallels with Tata Consultancy Services' MasterCraft tool [61]. MasterCraft is built using meta-object format (MOF), Query / View / Transformations (QVT), and Model-to-Text (MTT) template language. MOF and QVT are both OMG standards, and MTT is an in-progress OMG proposal. Clearwater, on the other hand, is based on XML technologies and therefore benefits from a large number of tools available that either operate on XML-based representations or produce XML documents. For example, ISG was constructed using the off-the-shelf XML packages Xerces and Xalan from the Apache project, and Ptolemy II exports an XML representation of workflows which is converted to XIP in order to provide GUI tool support for Infopipes.

2.4. Summary for the Clearwater Approach

Clearwater embodies an approach to building code generators that uses XML and XSLT with three prime characteristics: extensible domain specifications, pliable generators, and modular output. These three traits allow a code generator to resolve the three challenges of distributed heterogeneous systems presented in the introduction: abstraction mapping, interoperable heterogeneity, and flexible customization.

CHAPTER 3
THE ISG: CLEARWATER FOR INFOPIPES

3.1. Code Generation for Infopipes

One of the fundamental functions of operating systems (OS) is to provide a higher level of programming abstraction on top of hardware to application programmers. These abstractions may be implemented in multiple ways. For instance, block file access is abstracted as a byte stream through a software library. More generally, an important aspect of OS research is to create and provide increasingly higher levels of programming abstraction on top of existing abstractions. In particular, Remote Procedure Call (RPC) is a successful example of abstraction creation on top of messages, particularly for programmers of distributed client/server applications [10].

Despite its success, RPC has proven less than perfect for some applications. The primary reason for this is that it abstracts away too much information which the application often needs to run "correctly." This often results in the application developers re-writing code to discover the properties hidden by the RPC call. For instance, a streaming media application has distinct timing requirements, but an RPC call, by itself, contains no information about timing. Consequently, the writer of a streaming media application must add timers to his or her code to recover the "lost" timing information.

Infopipes, designed with information flow as a core abstraction, have been proposed as an appropriate programming paradigm for information-driven applications. Unsurprisingly, there is already some history of concrete examples of information flow software. For example, in UNIX combining filters yields a pipeline which is a precursor of the Infopipe programming style. This abstraction aims to offer the following

advantages over RPC: first, data parallelism among flows should be naturally supported; second, the specification and preservation of QoS properties should be included; and third, the implementation should scale with the application. This new abstraction is intended to complement RPC — not to replace it. In true client/server applications, RPC is still the natural solution

Even though RPC may not be a suitable abstraction for many applications, it has still contributed important tools for building distributed applications. Two of the most important are the Interface Description Language (IDL) and the RPC stub generator. Infopipes rely on a similar pairing of language and code generator for capturing and then mapping the system-level abstractions into executable code. Infopipe abstractions are captures in Infopipe Specification Languages (ISLs) and are converted into code by the Infopipes Stub Generator (ISG).

The main contribution of this chapter is the presentation and evaluation of current ISG and Infopipe implementations compared to existing middleware solutions for distributed programming. For performance comparison, microbenchmarks are used to show that Infopipe-based code achieves latency and bandwidth comparable to various existing communication software packages. For application evaluation, two application implementations showcase the utility of the ISG and AXpect weaver.

3.2. Lessons from RPC

The widespread use and acceptance of RPC has led to the development of higher-level architectural models for distributed system construction. For example, it is a cornerstone for models such as client/server (including Internet protocols like HTTP), DCOM, and CORBA. The client/server model is widely considered to be a good choice

for building practical distributed applications, particularly those using computation or backend database services.

On the other hand, several emerging classes of distributed applications are inherently information-driven. Instead of occasionally dispatching remote computations or using remote services, such information-driven systems transfer and process streams of information continuously (e.g., Continual Queries [64][66][68]). Applications such as electronic commerce combine heavy-duty information processing (e.g., the discovery and shopping phase involves querying a large amount of data from a variety of data sources [68]) with occasional remote computation (e.g., buying and updating credit card accounts as well as inventory databases). Even static web pages can conceal an information-driven flow of information since a single request for a page frequently generates both multiple requests to sites' backend software and also more HTTP requests by the client as it renders a web page.

The Infopipe abstraction was defined to cover some aspects of programming information flow applications unsupported by RPC. For example, many information flow applications involve multiple transformational stages and multiple communication links between the stages. A second example is the support for quality of service (QoS). Some existing efforts in supporting the specification and maintenance of sophisticated information flow applications using RPC-style middleware show the difficulties in such an approach [69].

On the other hand, while these models have proven successful in the construction of many distributed systems, RPC and message passing libraries offer limited support for information-driven applications because their natural operation does not reflect RPC's

request-response style. One example is bulk data transfers in which large numbers of requests and responses may be generated to cope with network reliability [42]. Another example is when information flows have quality of service (QoS) requirements, then certain elements of distribution transparency – an oft-cited advantage of RPC – can cause more problems than they solve. Various solutions to these shortcomings have been proposed. For example, the A/V Streams specification was appended to CORBA to addresses the deficiency of RPC in handling multimedia flows [AVStreams].

The IDL provides an easy way to specify the connection between two computation processes – the client and the server. It abstracts the connection and data typing and frees the programmer from the necessity of uncovering architecture-specific and even language-specific characteristics such as byte ordering or floating point representation. Given an IDL description, an RPC stub generator then implements the implicit functionality. Using it enhances program correctness and shortens development cycles, not just because it obviates the need for a developer to write large amounts of code, but also because the code it generates has already been tested and is far more likely to be "correct." Therefore, it greatly reduces both syntactic and logical errors appearing in communication code which in turn results in a great time reduction for application development.

Communication between processing stages in information flow applications inherently requires a large amount of communication code. Potentially, the code for each communication link must be tailored to the specific type of data exchanged. This code can consume a large amount of initial development time and debugging. Later changes are potentially difficult, as even a simple datatype change may require changes in several

program modules. Aside from data marshalling and unmarshalling problems, developers must also contend with finding services, and then creating and maintaining links to them.

The overhead for programming information flow applications manually is easily illustrated. For an information flow application that processes data in *n* serial stages, there are at least 2*(*n*-1) communication stubs. Any change in the data description carried over a single link, furthermore, presages a change in data marshalling code in two communication interfaces. Of course, some of these changes may require only a small change, as adding a simple extra integer to be propagated. Others may have a moderately more complex requirement, such as supporting a dynamic array. Then, there are changes which require extensive code writing and testing, as adding encryption or reliability code.

The overhead of this programming style has several consequences. Information flow systems built as point-to-point links in this way are not easily portable to new communication protocols, computing platforms, or languages. Due to the nature of the point-to-point development style, changes in each point-to-point link are likely to become brittle over time as more (link-specific) features are added and each communication link loses coherency with the whole system. The Infopipes abstraction alleviates this problem by avoiding point-to-point restrictions on specifications and incorporating communication in the abstract specification for the application.

3.3. The Infopipes Abstraction

Like RPC, Infopipes raise the level of abstraction for distributed systems programming and offer certain kinds of distribution transparency to support information-flow applications [59][86][100]. Example applications include data streaming and filtering [100], building sensor networks, e-commerce transactions and multimedia

streaming [86]. Infopipes were developed as part of the Infosphere project with the support of the DARPA PCES (Program Composition for Embedded Systems) project. Experiences from that project have shown that Infopipes can reduce the effort and code required to develop connections between the embedded imaging system of the UAV and the receiver of the image stream. Furthermore, Infopipes can employ various data management techniques to maximize communication quality.

A typical Infopipe has two ends – a consumer (*inport*) end and a producer (*outport*) end – and implements a unidirectional information flow from a single producer to a single consumer. Between the two ends a developer provides a function, the *middle*, which can process, buffer, filter, or transform information. In operation, an information producer exports and transmits an explicitly defined and typed information flow, which goes to a consumer Infopipe's inport. After appropriate transportation, storage, and processing, the information flows to a second information consumer.

Conceptually, an Infopipe is a mapping that transforms information units from its input domain to the output range between its consumer and producer ends. Infopipes go beyond the datatypes of RPC and specify the syntax, semantics, and quality of service (QoS) properties which are collected into a structure called a *typespec*. To close the gap between the abstract specifications and executable application code, developers need toolkits to developers Infopipe construction, composition, management of abstract data and QoS, and for incorporating the Infopipes into target applications which are increasing in complexity and heterogeneity.

3.4. The Infopipes Toolkit

The Infopipes toolkit was developed to support the implementation of Infopipes for information flow applications; the approach parallels the IDL and stub generator of RPC by introducing a specification document and code generator. Typespec information is captured in Spi (for Specifying Infopipes), a text-based Infopipe Specification Language, to describe the composition and connection of Infopipes in information flow applications. It is similar to other domain-specific languages such as Devil [73] or the RPC IDL in that it encapsulates domain knowledge and provides similar development benefits.

Besides abstracting the connection and data typing, a generator-based approach frees the programmer from the necessity of uncovering architecture-specific and language-specific details. Other advantages include enhanced program correctness and shorter development cycles since generated code is pre-written and tested. Infopipe specifications are translated by the ISG toolkit into code in a fashion analogous to an RPC stub generator thereby hiding from the developer the difficulties of marshalling and unmarshalling data, system initialization, etc.

The C/TCP implementation of Infopipes operates in a straightforward manner, transmitting bytes in native binary between the sender and receiver. Dynamic arrays and strings are only transmitted up to their used size as set during runtime. Datatypes as specified in Spi are translated into C `struct` statements. To avoid copying, there is one structure per port into which the port can write data (for an inport) or from which it can read data (for an outport).

After the specification language, the second part of the toolkit is a code generator, the Infopipe Stub Generator or ISG, which consumes the intermediate format generated

by an ISL and produces compilable source code in a fashion analogous to an RPC stub generator. The stub generator hides the technical difficulties of marshalling and unmarshalling data and manipulating system-specific mechanisms for property enforcement. The generator is the shield between the application developer and the complexity of heterogeneous operating systems, differing hardware platforms, and the translation from language-level abstractions to underlying message-based implementations.

The toolkit benefits the programmer in two ways:: first, a developer can easily move to a new underlying implementation as circumstances dictate since the abstraction encodes semantics about the application and not just implementation code. A second benefit is that the programmer can use multiple communication layers and connect together pipes written in multiple languages. In fact, a single pipe may use different underlying communication layers for each incoming or outgoing connection when connections to legacy software are involved. For instance, a pipe may receive data via a socket as its input, but it later may output that data using a second communication protocol such as HTTP. This type of situation is actually quite common, as the three-tiered architecture common for building web applications often uses HTTP for client/web server communication, and at least one other protocol (such as SQL via JDBC) for communication with backend machines.

Separating the ISL and the ISG, adds a level of flexibility. A standard intermediate format based on XML (the XIP) logically separates the language from the generator. Using an intermediate format as a connecting element allows the use of variant ISLs as input for generating Infopipe applications. This way, the second step (the

actual code generation) can be developed in parallel to the design and evolution of variant Infopipe Specification Languages. Furthermore, the intermediate specification opens the door for the ISG to support non-Infopipes flow-based languages such as Spidle [28] perhaps eventually allows developers access to capabilities from several research efforts.

At the top of the toolkit stack, Spi is a simple prototype language for describing Infopipes.

3.5. Spi: Specifying Infopipes

Programmers interact with the ISG through specification languages, and the ISG is designed to accommodate multiple specification languages each targeted to a different audience. Spi (Specifying Infopipes) is text based, and largely declarative with an eye towards for human-friendliness and a high degree of readability. The XIP intermediate language, on the other hand, primarily supports programmatic access to specification data and fulfills a need for a flexible specification format and will be covered in the next section.

Spi's primary purpose was as an experiment in creating a declarative Infopipes language. Given its narrow goals of Infopipe specification, Spi naturally fits in the regime of domain specific languages (DSL's). While it is common for DSL's to have provable properties, Spi's emphasis is on capturing of domain structures – i.e., Infopipe related structures – and their implementations for the time being.

For a developer, converting a Spi specification to compilable source code is a three-step process:

1. Create a Spi document to describe the information flow system.
2. Compile the Spi into XIP.

3. Compile the XIP to source code through the ISG.

Comparable to other languages, a context free grammar captures Spi syntax and it is compiled with a traditional lexer/parser suite, but instead of compiling direct to source, it is converted to XIP.

It is here easy to see drawbacks of performing code generation directly from Spi using a traditional lexer/parser approach. The result is a static language “locked in” to the recognized grammar. To extend Spi, one must update the entire lexer/parser package, from token recognition to intermediate representation, to output routines. This lack of extensibility leads to a curious cascade of complexity, which is one of the motivating factors for the current system. First, to implement a new feature for an Infopipe, it must be inserted into the code generation template facility. Then, to test, one must extend the specification language, the templates’ code, and the result code. Obviously, having to perform a revision of a language parser with every new research avenue proved a serious bottleneck to research progress. Therefore, Spi is only updated at comparatively long intervals as features in Infopipes and the Infopipes XIP documents become more “settled” in their implementation throughout the system.

It turns out that in the Infosphere project it was not unusual to have two or three developers working concurrently and semi-independently on Infopipes projects; each developer extended XIP to support his own explorations. The Spi/XIP disconnection deflects grammar conflicts and mitigates the risk of forking Spi into incompatible dialects.

For instance, mapping from Infopipe data types into native-language datatypes are handled in two different paths for code generation. C datatypes are encoded with the code

generation templates, but C++ datatypes, which were developed later, choose a more flexible scheme and allow embedded datatype mappings in the XIP specification.

3.6. XIP: XML for Infopipes

Spi, as discussed, is not desirable for direct code generation because of the inflexibility imposed by its grammar and consequential maintenance, but the question arises, why not use XIP alone? From a usability perspective, XIP's XML syntax significantly impedes readability and understandability of the specifications it encodes. While the syntax rules are simple for XML, the necessary proliferation of start/end tags and '<', '>', and '&' symbols can easily double the character count for a simple specification; yet, they signify more to the XML parser than they do to a specification developer. Using Spi alleviates this "visual burden" resulting in easily readable, easily writable specifications. Its subsequent compilation into XIP helps provide Infopipes both readability and extensibility.

Spi documents have three types of statements: *datatype*, *pipe*, and *compose*. Datatype is used to declare the contents of an application packet that is exchanged between two Infopipes. Typically, each Infopipe inport or outport will have a datatype assigned to it. It supports four basic types: float, integer, string, and byte. It also supports arrays over these basic types. Each datatype is given a name for reference by Infopipes.

A pipe, of course, describes either a simple or a complex Infopipe. A simple Infopipe is one that is not composed of smaller Infopipes, and a complex Infopipe, naturally, is constructed from the composition of other Infopipes. Each Infopipe, both simple and complex, can have inports or outports, and each inport or outport refers to a type previously specified via a datatype statement. For complex pipes, the developer must

specify the component Infopipes and how they are connected. The connections are declared by simply listing which pipe's inport port connects to another pipe's outport. Since a complex pipe does not "run," but is instead just a collection of simple pipes, if a complex pipe has an inport or outport it must describe the mapping of the complex pipes' ports to simple pipes' ports. This is accomplished by simply creating an "alias" directive which means that the complex pipe port name is an alias for a simple pipe's port name.

```
<pipe class="ImagePipelinePlain">
   <subpipes>
     <subpipe name="imagesSource"
              class="SendingPipe"/>
     <subpipe name="imagesReceive"
              class="ReceivingPipe"/>
   </subpipes>
   <connections>
     <connection comm="tcp">
       <from pipe="imagesSource"
             port="out"/>
       <to pipe="imagesReceive"
           port="in"/>
     </connection>
   </connections>
</pipe>
```

Figure 7. Example XIP Infopipe specification.

The compose declaration is a directive to the code generator to produce a pipe. It has only two parameters. First, the developer should provide the type of pipe to compose – one previously defined, of course, and the developer also provides the name the pipe is to be composed as this will govern its generation and organization on disk.

Strictly speaking XIP currently encodes more information than Spi, and as mentioned the flexibility benefit of XIP. XIP has three more advantages. First, the XIP is designed so that that specification compartmentalizes the various domain structures (pipes, datatypes, mappings, etc.) in differing XML elements. The ISG creates a data

repository as these fragments stored on for persistent, reusable specifications. Second, XIP is converted into XIP+ inside the code generator. While they are substantially similar, the flexibility of the XML format is key to supporting both formats. A third benefit manifests during code generation itself. Rather than emit bare source code, the ISG instead rewrites the XIP+ document with the generated code and thereby preserves information from the specification document that is usually discarded. This particular advantage is utilized by the AXpect weaver.

3.7. ISG: The Infopipes Stub Generator

The ISG is the oldest, most developed, and most refined of the three Clearwater generators presented in this dissertation. For this reason, it is presented in two parts. The following section presents the structure, implementation, and mechanics of the base generator. The "base generator" is defined as the templates which are executed to create only communication code for distributed Infopipes. From this discussion of the base generator follows a description of the AXpect weaver, which enables customization of the output code from the base generator in support of QoS goals.

3.7.1. The Base Generator

The ISG follows the general pattern of a Clearwater generator as already described in Chapter Chapter 2. As an application, the ISG consists of C++ pre-processing and post-processing stages around a core XSLT-based code generator. The pre-processing assembles a XIP+ document and handles repository interfacing, the XSLT generates code, of course, and the post-processor writes the results to disk. The ISG also has varying support for additional language and communication layer pairings with the

ISG. These include C++ using CORBA, local IPC, or local function calls, and Java and XML over TCP.

Figure 8 illustrates the stages of the ISG and AXpect weaver (described it more fully in the following sub-sections); corresponding source sizes. During generation, the specification AST is maintained as a DOM tree in-memory. Leaving discussion of the AXpect weaver for later, ISG code generation proceeds as follows:

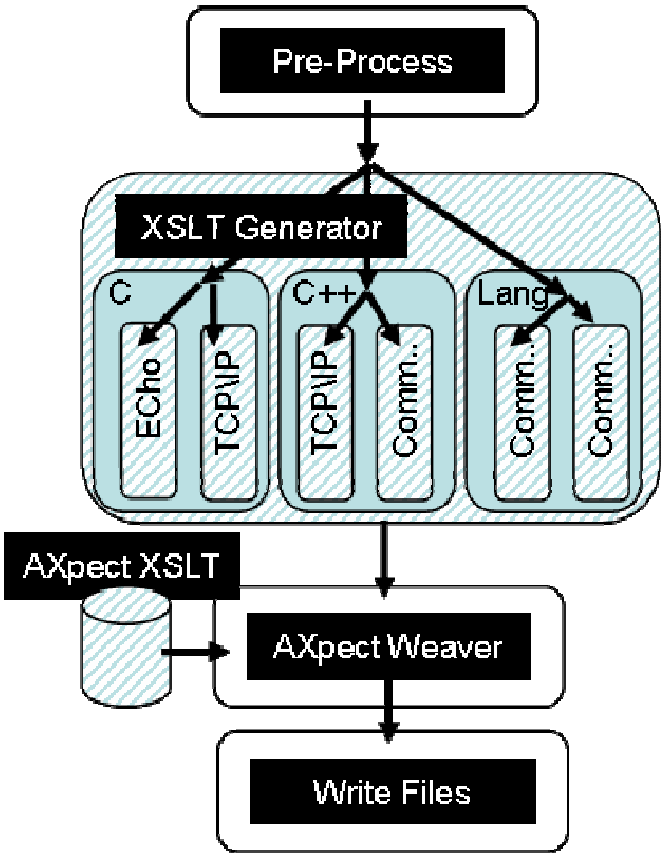


Figure 8. The ISG.

1. The Infopipe XIP description is divided into several sections of datatypes, pipes, filters, etc. and writes the specification fragments to the repository.

2. Elements designating which pipes to build are retrieved from the input XIP. Each forms the nucleus of a new document, termed XIP+, which is built from stored specifications and has verbose connection information.
3. The ISG passes the document to and invokes an XSLT processor to execute generation templates. Both the generated code and the reconstituted XIP+ are retained after code generation.
4. The specification+code is passed to the weaver.
5. Finally, XML markup is removed, and the code is deposited into files and directories, ready for use in an application.

At the top of the hierarchy, the ISG invokes a master template located in a well-known directory that run-time includes templates for each supported language. The ISG can load this template and apply it, via the XSLT processor, to the specification being processed. Various implementation language templates reside in separate directories. The next level of organization is to create sub-directories in each language directory for each communication platform available for that language.

In Figure 9, the C subdirectory has templates for generation of C core code, C runtime support, and a map table for mapping Infopipe specific data primitives to C types. Within the C subdirectory there is a TCP subdirectory that contains the XSLT templates for implementing TCP connections between Infopipes. Likewise, ECho for C has a parallel subdirectory and allows the two communications implementations to share the core code. Likewise, there is a CPP (C++) subdirectory for C++ generation templates with it, too, having multiple communication language subdirectories.

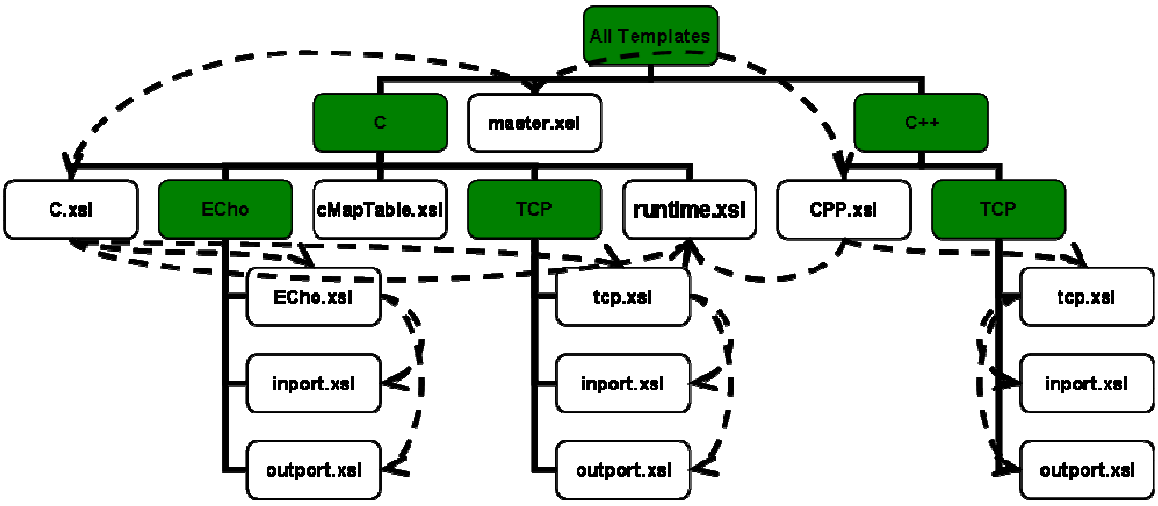


Figure 9. XSLT template organization for C and C++ TCP Infopipes.

The ISG output for the Infopipes C implementation follows the traditional approach of dividing code into file-level modules and each file corresponds to one functional unit of an Infopipe. The generated C++ implementation follows an object-oriented decomposition into base classes and subclasses corresponding to functional units. Despite this, the two implementations can have shared code. For instance, C++ directly generates using C runtime support templates code for publishing and discovering Infopipe connection information. Other times the generators have structural similarities, such as in unmarshalling code, but due to language differences are not shared, *e.g.*, unmarshalling data to a `struct` for C but a `class` for C++.

One of the most important goals of the Infosphere project is addressing quality of service, such as data latency, security, or resource control, for information flow systems. However, the basic code generation capability of the ISG does not implement generation of any code to support QoS. Furthermore, it seemed that quality of service were explicitly designed into the generator, it would be difficult to anticipate all possible QoS scenarios as QoS is likely to vary from application to application not only by parameters (e.g.,

10ms latency versus 10s latency) but by feature mix as well (e.g. latency controlled versus security controlled).

In light of this, an aspect-oriented approach to QoS appeared warranted by which the generator output could be customized. Unfortunately, while there are several Java aspect weavers, there are no well-supported or widely used weavers for C or C++, the primary target languages. Still, some projects had been successful at marrying DSL techniques and AOP [7]. ISG development in this space produced the AXpect weaver which experiments have shown to be useful in controlling QoS and implementing web service level agreements [102], and that the approach encourages good reuse of QoS code [104].

3.7.2. The AXpect Weaver

The AXpect weaver is the component of the ISG that brings together the preceding three topics by interpreting aspect specification statements in the XIP, loading the aspects from disk, and applying them to the template-generated code. The modular structure of the ISG made it easy to insert the AXpect engine as a processing stage executed after applying the XSLT code generation templates, as shown in Figure 8.

AOP has three types of advice: before, after, and around. An application developer chooses a joinpoint using a pointcut, and then designates by keyword whether aspect code should execute before, after, or around (which subsumes instead-of) the selected joinpoint. One interesting subtlety is that in AXpect the explicit XML tags denote a semantic block of code, and not just a single point. This most closely relates to AspectJ “around” semantics, but in practice retains the before and after capability of the weaving, without loss of “power.” One could also view it another way, in which the

XML opening tag marks “before,” the closing tag marks “after,” and taken together the tags make up “around” semantics.

In the ISG, there is only aspect support at the XIP level, and the research question of a language for aspect specification in Spi is still open. The decision stems from two main reasons. First, XIP-level aspect specification is required in any case, since Spi is translated into XIP. Second, there is no standard WSLA specification language – competing standards include CDL from the QuO project [69], and proposals by HP [93] and IBM [33].

There are three key concepts that enable the weaver. First, semantic tags in the generator denote important regions in the generated source code, the joinpoints. Second, XSLT is used to implement the mechanics of the weaving of the aspect, making pointcuts and weaving advice, and third, the process is governed by weaving directives inserted into the Infopipes description file.

3.7.2.1. Joinpoints: Aspect Support in the Templates

Any weaver must have some points in the target code that it can identify. These are the “joinpoints” which are similar to annotation joinpoints described by Kiczales [ECOOP05]. In Clearwater, generators benefit from the domain specific nature of the problems for which they are designed. Because of this, the generator developer knows that specific and well-defined sets of activities occur within each Infopipe with known ordering. For example, an ISG developer knows that each pipe has a start-up phase that includes starting up each inport and each outport, resolving bindings and names, and actually making connections. During these initializations, the Infopipe may initialize data

structures for each inport or outport. In the code generation templates, there is template code for each of these “common tasks.”

For a concrete example, consider a fragment of template for generating C code Infopipes. This template excerpt generates a startup function for the Infopipe. The startup function name is based on the name of the Infopipe. The XSL commands are XML tags which have the `xsl` namespace prefix (like the element `xsl:value-of` which retrieves the string representation of some XML element, attribute, or XSLT variable). The added joinpoint XML tag is bolded, and it denotes the beginning and ending of the code block that implements the startup functionality. Reverse-printed text denotes the joinpoint XML for clarity, and the C code is printed in bold to distinguish it from the XSLT.

```
// startup all our connections
int infopipe_<xsl:value-of select="$thisPipeName"/>_startup()
{
  // insert signal handler startup here
  <jpt:pipe point="startup">
  // start outgoing ports <xsl:for-each select="./ports/outport">
  infopipe_<xsl:value-of select="@name"/>_startup();</xsl:for-each>
  . . .
  </jpt:pipe>
  return 0;
}
```

Figure 10. Example generator template code with joinpoints.

Sometimes a joinpoint does not denote executable code but language-specific structures that are needed for code to be written correctly. In following example, the joinpoint denotes the header file for an inport. This allows the C developer of new aspect code to insert new function definitions at the appropriate scope.

```
#ifndef INFOPIPE<xsl:value-of select="$thisPortName"/>INCLUDED
#define INFOPIPE<xsl:value-of select="$thisPortName"/>INCLUDED

<jpt:header point="inport"  pipename="{$thisPipeName}"
                            portname="{$thisPortName}">
int drive();
// init function
int infopipe_<xsl:value-of select="$thisPortName"/>_startup();
int infopipe_<xsl:value-of select="$thisPortName"/>_shutdown();
void infopipe_<xsl:value-of select="$thisPortName"/>_receiveloop();
// data comes in to this struct
extern <xsl:value-of select="$thisPortType"/>Struct
          <xsl:value-of select="$thisPortName"/>;
. . .
</jpt:header>
#endif // Infopipe<xsl:value-of select="$thisPortName"/>INCLUDED
```

Figure 11. Example generator template code with language-specific joinpoints.

Joinpoints remain with the code until the files are written to disk. After the generation phase, they serve as signposts to the weaver and aspects. Returning to the first joinpoint example, then after generation for the pipe called "`imageReceiver`" there is this startup code (all XSLT has been evaluated and removed):

```
// startup all our connections
int infopipe_imageReceiver_startup()
{
  <jpt:pipe point="startup">
  infopipe_inp_startup();
  infopipe_inp_receiveloop();
  </jpt:pipe>
  return 0;
}
```

Figure 12. Output with XSLT evaluated and removed, but joinpoints retained.

At this point, obviously, the code looks very much like pure C ready for compilation, but most importantly, the aspect writer and the AXpect weaver know what the code does in the context of the Infopipes domain. Interestingly, so far only about 37 joinpoints are necessary for quite a bit of flexibility with respect to actions that can

perform on the generated code. These joinpoints have evolved into some broad categories as evidenced in Table 1 and Table 2.

Table 1. Infopipes domain joinpoints.

Data Typing	Pipe	Communication	Data Typing
data:define data:initialize	pipe:startup pipe:shutdown	comm-startup comm-shutdown	data:define data:initialize
Inport	**Outport**	**Middle**	**Inport**
inport:startup inport:read inport:unmarshall inport:callmiddle inport:shutdown	outport:marshall outport:push outport:startup outport:shutdown	middle:startup middle:shutdown middle:userdeclare middle:userfunction	inport:startup inport:read inport:unmarshall inport:callmiddle inport:shutdown

Table 2. Target specific joinpoints on Infopipes.

C/C++	C++ only	make
header:pipe source:pipe header:inport source:inport header:outport source:outport source:userdeclare	class:pipe class:inport class:inport	make:rule make:link-exe

C++ TCP	C ECho
socket:socket socket:bind	echo:header echo:module echo:startup echo:shutdown

The first group of joinpoints relates the code directly to the Infopipes domain. In effect, each joinpoint maps the code back up to the context of higher-level abstractions. Currently these joinpoints are grouped first by the abstract structure they annotate, and secondarily by the functionality the code implements.

Target specific joinpoints, on the other hand, help a developer structure his code properly for the proper output target. Languages have syntactic conventions that must be obeyed, and these joinpoints make the language syntax clear to a coarse granularity. For instance, in C variable declarations occur when a new scope is opened, as at the beginning of a function. "`source:userdeclare`" is the point in the source code where an aspect writer can weave in additional variables that support the implementation of the aspect.

Of course, the generated document is still a XIP+ document. Therefore, being XML, the natural method for interaction with a XIP+ document is to use XSLT and XPath.

3.7.2.2. Pointcuts and Advice: Implementing an Aspect

The second ingredient of the AXpect weaver is an XSLT file that contains aspect code. Every AXpect file has two parts. First, the aspect has some pattern matching statement, written using XPath and utilizing the XSLT pattern matching engine, to find the proper joinpoint and the code to be inserted. The pattern specification role corresponds to the role of pointcut patterns in an AOP system like AspectJ. Instead of regular expressions and language specific expressions, however, the pointcut in AXpect is an XPath predicate for an XSLT match statement like this:

```
//filledTemplate[@name=$pipename][@inside=$inside]
                                  //jpt:pipe[@point='shutdown']
```

By dissecting the elements of the pointcut XPath statement it is possible to understand its function:

`//filledTemplate[@name=$pipename][@inside=$inside]` – structure-shy specification to find a `filledElement` template, which is a block of generated code and predicates to narrow filled templates returned to one for a particular pipe.

`//jpt:pipe[@point='shutdown']` – a specific joinpoint

Instead of keywords like AspectJ, the AXpect developer uses placement. The actual joinpoint and its contents are copied over by XSLT's `xsl:copy` instruction. A simple aspect for AXpect looks like this (the C code is bolded for distinction from the XSLT):

```
<xsl:template match="//filledTemplate[@name=$pipename]
          [@inside=$inside]//jpt:pipe[@point='shutdown']">
  fclose(afile);
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
</xsl:template>
```

Figure 13. A simple aspect in XSLT for AXpect.

It is now easy to see how aspect code, pointcuts and joinpoints, and advice mesh. The pointcut, in reverse print, is contained in the `match` attribute to the `xsl:template` element. The C code to close a file (`fclose(afile)`) is located before the `xsl:copy` command, which means that it will be executed before the rest of the shutdown code. The `xsl:apply-templates` is boilerplate XSLT that ensures the processor continues to pattern match to all elements and attributes of the generated

document that lie inside the joinpoint element. (Future plans, in fact, are to eliminate having to write XSLT for aspects, and the accompanying boilerplate like the `xsl:copy` elements and to generate them from a higher level description.)

One of the interesting results of using XSLT and XML for this system is that aspects can introduce new joinpoints in the form of new XML tags. . The denotation is accomplished in the same fashion as adding joinpoints to the original templates -- by adding XML elements to the aspect template in the same manner as XML tags were added to the code generation templates. This means that one aspect can build upon an aspect that was woven into the code earlier (order of aspect weaving will be discussed shortly). In the example scenario, as aspect inserts timing code to measure how long various pieces of Infopipe code take to run the user function which can be used later in calculating CPU usage.

```
<xsl:template match="//filledTemplate
                      [@name=$pipename][@inside=$inside]//jpt:inport">
  <jpt:time-probe point="begin">
  // take timing here
  gettimeofday(&amp;inport_<xsl:value-of select="@point"/>_begin,NULL);
  </jpt:time-probe>
  <xsl:copy>
    <xsl:apply-templates select="@*|node()"/>
  </xsl:copy>
  <jpt:time-probe point="end">
  gettimeofday(&amp;inport_<xsl:value-of select="@point"/>_end,NULL);
  </jpt:time-probe>
</xsl:template>
```

Figure 14. An excerpt from aspect that introduces joinpoints.

The timing code is bracketed with XML that declares it, and the CPU monitoring code can then select it with a pointcut just like any other joinpoint:

```
<xsl:template match="//filledTemplate[@name=$pipename][@inside=$inside]
                     //jpt:inport[@point='callmiddle']
```

```
//jpt:time-probe[@point='end' ">
```

An aspect may also need to refer to data in the XIP specification. Since the specification is presented along with the code for weaving, the aspect code can refer to the specification in a same manner that templates retrieve the data by using `<xsl:value-of>` element.

This also holds true for data that must be retrieved from external documents, as from a WSLA that captures a QoS specification. XSLT provides a "`document()`" function which allows for a developer to integrate data from any XML source document.

The last piece of the aspect weaving puzzle is the process of bringing aspects together with the base code.

3.7.2.3. Weaving: AXpect Execution

One of the benefits of an extensible specification is that not only can new domain-specific statements be added to the specification, but new generator directives can be added as well. AXpect takes advantage of this feature weaving is controlled by the introduction of new statements which are captured after generation by the AXpect weaver module of the ISG.

Adding aspect statements to each pipe specification that generates code is done by adding an XML element which carries the name of the aspect and any additional information the aspect requires. For instance, if to apply an aspect to the receiver that generates rate controller functionality and it references a WSLA it is simply:

```
<apply-aspect name="rateController.xsl" doc="uav.xml"/>.
```

Aspects may specifically rely on functionality located in other aspects. Naturally, such a dependency will imply at least a partial ordering. Developers can denote this in the

specification by nesting aspect application elements within one another, and the weaver will apply the most deeply nested aspects first. For instance, if a Unix application requires a CPU usage monitor, the implementation will rely on timing information. In this case, the timing information can be generated by weaving in the timing aspect. In the declaration of Figure 15, `rate_controller` depends on two aspects, and one of those, `cpumon`, depends further on `timing` and `sla_receiver`.

```
<pipe class="vidSink" lang="C">
  <apply-aspect name="rate_controller.xsl" targetPct="20">
    <apply-aspect name="control_receiver.xsl" target="ppmIn"/>
    <apply-aspect name="cpumon.xsl" target="ppmIn">
      <apply-aspect name="timing.xsl"/>
      <apply-aspect name="sla_receiver.xsl" doc="uav.xml"/>
    </apply-aspect>
  </apply-aspect>
  <ports>
    <inport name="ppmIn" type="ppm"/>
  </ports>
</pipe>
```

Figure 15. `apply-aspect` statements within XIP.

The AXpect weaver does not require a developer to specify dependencies between all the aspects in use – aspects that are at the same "dependency level" are simply woven as they are encountered in the document. Consequently, for the Infopipe XIP in Figure 15, the order of weaving will be: `timing`, `sla_receiver`, `cpumon`, `control_receiver`, and finally, `rate_controller`.

Since this nesting implies that aspect weaving requires an indefinite number of invocations of the XSLT processor, and neither the XSLT standard nor Xalan-C supports self-invocation, the evaluation of these statements is handled in a C++ program using

Xerces-C, which is the platform the ISG is built around. The weaver proceeds recursively through the following steps on each pipe:

1. Retrieves the first `<apply-aspect>` element from the pipe specification.
2. If the element contains more `<apply-aspect>` statements (which are dependencies), then the AXpect applies those aspects first, and re-enters the process of weaving at this step.
3. The weaver retrieves the aspect code from disk (aspects are kept in a well-known directory).
4. Apply the aspect to the code by passing the aspect XSLT stylesheet, the generated code with joinpoints, and system XML specification to the Xalan-C XSLT processor. The result is a new XIP+ document that again contains the specification, woven code, and joinpoints. The weaving result serves as input for any aspects that follow the current aspect. This includes aspects which depend on the current aspect's functionality, or functionally independent aspects that are simply applied later.
5. Apply the next sibling aspect, entering at step 2.

Once all aspects are applied, the entire XML result document is passed to the last stage of the generator. Any residual XML joinpoints in the woven code remain until the last stage removes them as the code the generator writes the source files to disk.

This algorithm implementation only required an additional 79 lines of C++ code be added to the generator application. The bulk of the weaver complexity is contained by the XSLT library that performs weaving.

3.8. Benchmark Comparisons

Three performance metrics commonly concern information streaming applications: latency, throughput, and jitter. To gauge the overhead and relative performance of Infopipes to other approaches providing communication links, ISG code is compared is compared in performance over several microbenchmarks for these metrics with SunRPC, CORBA, and hand-written communication code.

Given that information flow applications often involve more than a single point-to-point connection, it is necessary for to assess the communication software with more than a single point-to-point connection. These benchmarks involve two computers, one a source and one a sink, and in benchmarks that involve three computers, in which case the third computer is interposed between the sink and source. Therefore, the benchmarks test two different software configurations: 2-node and 3-node. Given these differences, the tests provide latency, jitter, and throughput metrics for both configurations.

3.8.1. Communication Software

A benchmark against SunRPC is useful for several reasons. First, it is common; it ships as part of most Linux and Unix distributions. Second, it is in common use; this protocol and tool provides the underpinnings of many NFS (networked file systems). Third, given its duration of use and widespread adoption, there should have been ample opportunity for the community to optimize its generated code for performance. In fact, SunRPC functions are part of standard GNU C library. The generated functions handle a variety of tasks including name resolution, marshalling and unmarshalling to a standard format known as XDR (the external data representation), and semantics enforcement. SunRPC supports abstraction of communications via an interface definition file in which

data types and function names are defined. This file is processed by a code generator to create code skeletons supporting data transfer and the execution of remote procedures.

While most decisions in SunRPC are made by the generator on behalf of the application programmer, the programmer must choose a return type and a network-level protocol for transmitting the data. Return types are chosen in the IDL and result in different generated code. Choice of protocol, on the other hand, is made at run-time at which point the programmer can choose either UDP or TCP. Such a choice must be made carefully, however, as RPC over UDP does not support large data types.

CORBA is a more recent, object-oriented version of RPC. CORBA and its associated standards are governed by the Object Management Group, a consortium of businesses with the goal of creating standards that support interoperable enterprise software. One of its chief benefits is that CORBA was designed around object-oriented programming and therefore reflects the semantics of popular languages like C++ and Java. Furthermore, CORBA provides a great deal many more services than are provided with SunRPC. Lastly, a great deal of systems and software-engineering community research has focused on CORBA. In particular, the ACE+TAO project has emphasized building CORBA for real-time application support. It is comprised of two parts: ACE, the Adaptive Communication Environment, and TAO, The ACE Orb.

Like SunRPC, CORBA has similar semantics for computation and function calls. CORBA supports an IDL language closely related to the SunRPC IDL specifications but adapted to support object-oriented issues such as polymorphic types. CORBA functions are encapsulated in an ORB, or Object Request Broker, a library that intercepts and

routes inter-object communication to the proper functions. In ACE+TAO, the ACE components add support for real-time control and monitoring of quality of service.

Recently, the OMG has recognized the shortcomings in CORBA with respect to supporting information flows. This has lead to the A/V Streams specification. Unfortunately, this specification supports only the control of information flows and not the actual transfer of data in the flow. Programmers using the A/V Streams interface must still create the code for data transfers and connections manually. More recently, OMG has extended the model for CORBA computing to include “components.” The component abstraction, as a rough description, supports the aggregation of multiple CORBA objects inside a container. While the objects within containers still do not support streams, the containers themselves can support streams of information as sets of publishers (sources) and subscribers (sinks) to recurring events. However, implementation of this standard is still in its early stages.

Finally, Infopipes are benchmarked against hand-written code. Hand-written code can be optimized to the particular application since the programmer will have intimate knowledge of the behavior and needs of the application. However, the optimizations possible with a hand-written code carry with them two significant programming pitfalls. First, the hand-written code is likely to contain bugs. The elimination of bugs by using well-established code-bases is one of the key drivers towards library and generated communication packages. Second, hand-written code is likely to be difficult to maintain and evolve particularly if multiple developers are to add to the code over the life of the project. This can lead, again, to a preponderance of bugs as side effects accumulate with each change to the code base.

Furthermore, even while it is possible to optimize sockets for an application, it is not necessarily straightforward. Each protocol involved in the communication link will likely have multiple valid options, and the effects of these options may be difficult to discern. Various protocol options, such as TCP's "quick ACK" and the "Nagle algorithm" may be difficult to place in force, owing to non-obvious semantics, and even more difficult to maintain as an application evolves or as conditions change.

In each of these communication layers and in the case of Infopipes, the chief interested is the effect of communication software choice on some key metrics. Specifically, the interesting question is whether Infopipes' generated code can achieve performance levels comparable to hand-written code to the well-established RPC packages.

3.8.2. Metrics of Interest

There are three key metrics chosen for evaluating communication software because these metrics are directly relevant to information flow applications – latency, the time it takes for a task to be completed, jitter, a metric derived from latency, and throughput, the ability of the communication software to transmit data en masse.

Latency is the overall measure of how fresh (or, conversely, stale) the information provided is. Latency is a particularly interesting metric when a consumer of the information flow must act on the information arriving in a timely fashion. For instance, reconnaissance data from an aerial drone or real time stock quotes would have stringent latency requirements.

For these benchmarks, the measure of latency is the time elapsed from calling the communication software's function to send an application packet until receiving some

verification of receipt by the data sink. This is an end-to-end latency with one return value. For a two node latency benchmark, in effect, this scheme is effectively the round-trip latency. On the other hand, for a three node benchmark, this latency is the end-to-end latency plus the time needed for the acknowledgement. Data receipt verification is by one of two methods depending on the semantics of the communication software; either the data source sends the data and waits for a returned acknowledgement on a socket, or the RPC-style function returns. In the case of the latency tests the connection between sender and receiver is symmetric in available bandwidth. There are two reasons for obtaining latency measurements in this way. First, this method avoids clock drift between two computers over the course of the tests. Second, and more importantly, RPC-type calls, which "normal" operation for SunRPC and CORBA, are always two-way, request-response calls, and will always be subject to round-trip latency.

In networked applications, latency will appear from two sources: the software at each compute node responsible for interaction with the network, and in the network itself. Network latency generally lies beyond the control of applications programmers. In fact, network latency in some situations can be dominated by real-world distance in which the speed of light imposes a lower latency limit, and by virtual distance, if the data must make multiple network "jumps" to reach its destination. Of course, in various overlay architectures, communications software may be afforded a degree of control over the network routing mechanisms. For the purposes of these benchmarks, this is considered a side issue, and in fact research is ongoing into strategies for determining application level packet routing within an overlay network – a non-trivial problem.

Therefore, the tests are designed to assess latency as it is affected by a programmer's choice of communication software. Latency can be significantly affected by decisions made inside these frameworks such as whether or not data should be marshaled into a common format before sending. In rich frameworks, it may be related to large numbers of functions calls. Also, latency may stem from inefficient programming – e.g. multiple calls into the kernel each of which will prompt an expensive context switch for the CPU or excessive data copying.

Jitter is related to latency. Conceptually, it is the degree of uncertainty in the latency as measured from packet to packet. That is, jitter describes how different latency might be between each application packet's latency and the average latency for all packets. The metric is most important to information flow applications which need "smooth" flows of data. Typically these are media applications which attempt to deliver a continuous flow of images or sounds to an observer, as when someone watches a movie. For these μ-benchmarks, jitter is measured as the standard deviation of a set of (round-trip) latency measurements.

Jitter and latency may be traded off against each other in many cases. This is done by inserting either a buffer or a synchronization mechanism. In doing so, latency will typically increase – a buffer imposes extra time overhead as the data migrates through it, and a synchronization construct imposes extra time overhead as processes or threads block to wait on peers. Even though latency is increased, the programmer now has control over the program element introducing the latency. A buffer affords the opportunity to "soak up" jitter by providing a pool of immediately available elements. These tradeoffs are specific to applications, however, and even to tasks within

applications. A programmer will likely choose different latency/jitter tradeoff schemes for audio and video.

Throughput is also important – especially for applications with large data payloads such as scientific applications, high-volume business applications, and video streams. Throughput is measured as bytes per second by measuring the time taken to send many application packets from a source to a sink and then calculating based on the total volume of application data bytes sent. Defining throughput in this fashion eases the comparison of throughput across the differing types of application packets to assess the relationship of payload size to overhead.

Furthermore, the distinction between total bytes and total application data bytes highlights how the choice of communication software may layer in additional data to be transmitted in parallel with the application data. Furthermore, the maximum application-data throughput achieved by the communication software will indirectly reflect the overhead of the communication software by occupying some of the bandwidth. There are cases, however, in which total bandwidth must be watched closely. For instance, a mobile device may demand stingy communication software to minimize power drain during data transmission. In effect, to be most useful, a complete set of throughput benchmarks should define an application and its environment.

Such application dependence highlights a performance issue with respect to throughput. While there is a tradeoff between jitter and latency, managing throughput requires different, often application- or data-specific tradeoffs. For instance, a stream of images at 30 images/second might be degraded to lower quality if not enough bandwidth is available to send them at full resolution. On the other hand, depending on the

application, it might make more sense to send the pictures at a lower rate but at full quality – say 15 images/second. Sometimes, too, the application may need to direct which tradeoff is appropriate. For instance, if a UAV is streaming images back to a command post over a limited-bandwidth connection, it may send degraded quality images at full rate during flight to its station to ensure responsive maneuvering, but send lower rate, higher-quality images when on-station so they can be analyzed.

3.8.3. Benchmark Environment

The benchmarks were run on three computers attached to the Georgia Institute of Technology College of Computing's outland network. This allowed for "root" privileges for installing and using low-level profile tools as well as insulating the benchmarks from external traffic, and it also insulates the rest of the systems network from the traffic caused by the tests.

While two different kinds of computers were used in the test, each computer was installed with Debian distribution of Linux and a 2.6 kernel version. Care was taken to ensure that tests were consistent in their use of the computers. For instance, data sources were always assigned to "narita," data sinks were always assigned to heracles, and "midNode," the middle node for the three-node performance tests, were assigned to perseus. Note that while the Linux kernel was compiled using 3.3 series gcc for each machine, the tests were run using 3.4 series gcc, discussed in more detail below. Table Y provides a summary of each machine's capabilities.

Table 3. Machine configurations for benchmarking

	Narita	Perseus	Heracles
CPU	Pentium 4, 3.0 GHz	Pentium 4, 2.8 GHz	Pentium 4, 2.8 GHz
Memory	1 GB	512 MB	512 MB
Network interface	Intel 82547EI Gigabit Ethernet Controller (LOM)	Intel 82540EM Gigabit Ethernet	Intel 82540EM Gigabit Ethernet
Linux version	2.6.14 compiled with gcc 3.3.5	2.6.8 compiled with gcc 3.3.5	2.6.8 compiled with gcc 3.3.5
MTU	1500	1500	1500

For software, all libraries were compiled from source with the exception of the SunRPC code, which is included in the libc distribution. ACE+TAO was compiled with debugging off (default is debugging support turned on). All test code was compiled with –O3 optimization. While –O2 optimizations can yield higher-performance code in some cases, this can not be determined a priori and, in fact, may be dependent on specific machine and microprocessor architecture decisions (e.g., amount of on-chip cache).

The SunRPC package used was the RPC included with the libc version 2.3.5. The ACE version, which is the foundation for TAO, was 5.4.8, and TAO version built on top of ACE was version 1.4.8.

3.8.4. Benchmark Execution

Machines participating in benchmarks were controlled automatically via shell scripts and `ssh`. Each combination of software, software parameters, packet size, and metric of interest was first executed as a "dummy" run. This allowed the operating systems on the machines in question to work with warm caches so that benchmarks would not be skewed by a trial in which time was spent waiting for code pages to be fetched from disk. Once the cache was warm, each a series of fifty trials was run and the

metrics written by the executing program itself to a file. (Of course, these data files were wiped in between the cache-warming run and the actual benchmark.) Following the completion of the fifty trials, the metrics file was moved to a central location.

For post-processing of the metrics, data files were processed using a Python script to extract metric averages using a publicly available statistics package. The aggregated metrics averages could then be imported into a charting program for visualization.

3.8.5. Benchmark Transfer Data

The benchmark suite includes four different kinds of data types to be transferred: small, mixed, large, and image data packets. The first three data packets comprise the fully-synthetic set of benchmarks.

A small data type consists of a single 4-byte integer. The small memory footprint of the data type allows the benchmark to assess the overhead for the communication software, and not the overhead for data transfer.

Also tested was a relatively small “mixed” data type in which there are multiple types of data that must be transferred. For the benchmarks, the mixed data packet contained an array of 100 characters, a 10 element float array, and 3 integers (152 bytes total). If substantial overhead is present for marshalling the data for communication, then each of these data will exercise a section of marshalling and un-marshalling code.

The benchmarks included a “large” packet containing an array of 3,072 integers (12288 bytes). This is several times the network maximum transfer unit of 1500 bytes. Such large payloads maximize data throughput while minimizing the number of function and system calls needed to transfer the data.

A fourth packet type is the application-based benchmark, based on the UAV scenario.

The application benchmark involves the transfer of an image bitmap; the 427x640, 24-bit color image is representative of the type of data used in the UAVDemo application – an uncompressed image bitmap. Its total space, including a small amount of metadata such as image size and encoding, is 819,854 bytes. For the three-node test, this image was converted into a grayscale format at the middle node so that the first transfer was of a full-size, full-color image and the second image transfer was of a full-size, grayscale image.

As a whole, this selection of benchmarks provides a range of datatypes from simple octets to arrays of typed data which gives clues as to type-specific communication software overhead in both latency and throughput. It also includes a range of sizes over nearly six orders of magnitude which helps sift size-dependent overhead from overhead entirely related to the communication software itself.

3.8.6. Two Node Synthetic Benchmarks

The synthetic benchmarks were designed to test the communication software for their impacts on the metrics defined earlier. The benchmark is divided into two separate groups: benchmarks for two nodes and benchmarks for three node systems. Two node benchmarks, discussed in this section, are useful for clarifying the exact impact that the communication software has on the metrics. Because effects are largely confined to only the two machines participating in the test, detailed analysis becomes feasible. Three node benchmarks illuminate how impact can sometimes be cumulative across more actors multiple steps in information flow applications.

The first discussion of latency between two nodes serves as a basis for later observations on performance. The simplicity of the two node setup is conducive to profiling and careful analysis. Furthermore, both jitter and throughput, introduced after latency, are related to latency.

3.8.6.1. Latency

This test assesses latency for each of the communication software choices and for each of the application packet types as described above. In some cases, the tests include variations that are available to application programmers to alter the behavior they expect from the communication software. In general, Table 4 and Figure 16 show that the Infopipes implementations compared favorably to the other communications software across the tested range of parameters and packet types of small, mixed, and large. Some interesting observations may be made from these tests.

Reported in Table 4 are the averages of 30 trials. Each trial consisted of recording a time stamp, sending data, awaiting response from the server, and finally recording a time stamp. The number of application data packets sent per trial was 10,000 for the Small packets and 1000 for the Mixed and Large packets.

As a general trend exhibited by all tests, latency generally increases as application packet sizes get larger. There are two exceptions: the case of hand-written code, and the case in which Infopipes ECho latency drops from the small packet case to the mixed packet case.

Table 4. Results of latency benchmark for Small, Mixed, and Large application packet sizes.

Communication Software				Trials	Avg. Latency (ms)	Std. Dev. of Avgs. (ms)
Small	Hand	C	Default	30	0.496	6.60E-06
			QuickACK	30	0.243	0.001
			CloseListen	30		
	Infopipe	C	ECho	30	0.323	0.121
			TCP	30	0.140	0.040
		C++	TCP	30	0.121	0.0002
	SunRPC	int	TCP	30	0.258	0.118
		void	TCP	30	0.183	0.095
		int	UDP	30	0.216	0.080
		void	UDP	30	0.170	0.055
	TAO		twoway	30	0.256	0.009
Mixed	Hand	C	TCP	30	0.194	0.007
	Infopipe	C	ECho	30	0.292	0.126
			TCP	30	0.161	0.055
		C++	TCP	30	0.133	0.031
	SunRPC	TCP	int	30	0.351	0.090
		UDP	int	30	0.206	0.023
	TAO	C++	twoway	30	0.312	0.017
Large	Hand	TCP		30	0.326	0.001
	Infopipe	C	ECho	30	0.402	0.046
			TCP	30	0.299	0.054
		C++	TCP	30	0.266	0.027
	SunRPC	TCP	int	30	0.489	0.026
			void	30	0.485	0.029
	TAO	C++	twoway	30	0.398	0.004

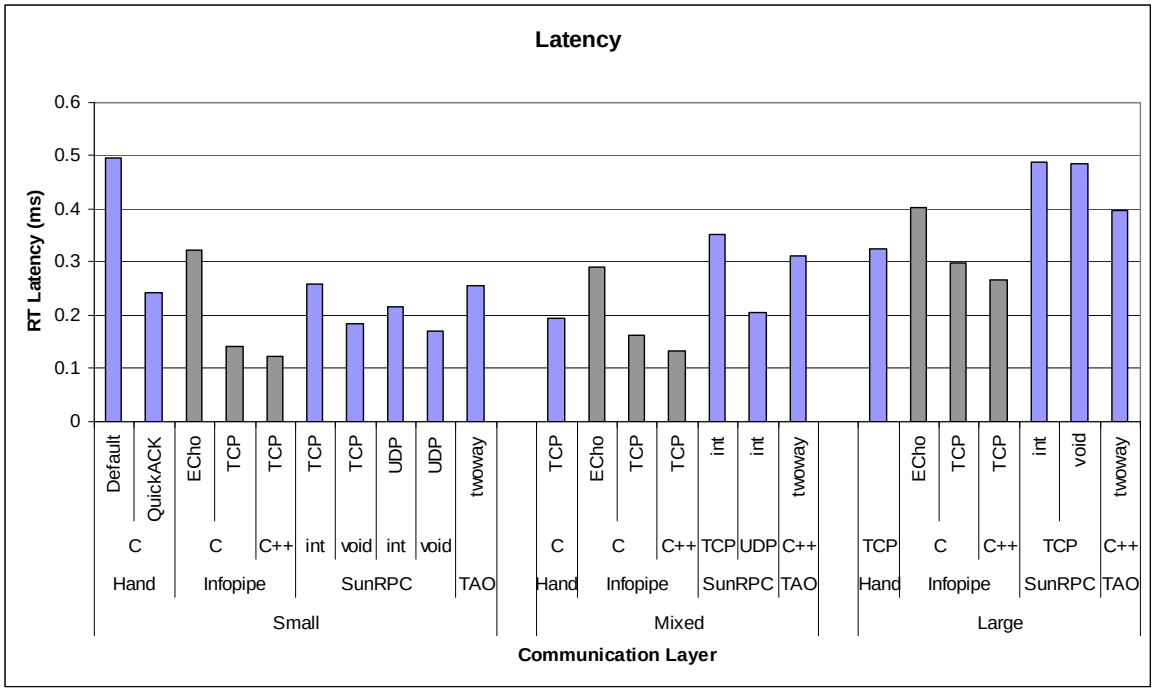


Figure 16. Synthetic benchmark latency results compared for Small, Mixed, and Large.

First, consider the Infopipes ECho anomaly first. The counterintuitive measurements raises the question of whether this is a sampling anomaly or whether the two average latencies really do differ. Since there is no clear trend in the values for Infopipes ECho, a statistical test of hypotheses is in order:

H_0: ECho Infopipes for Mixed and Small have the same average latency.

H_1: ECho Infopipes have a different average latency for Mixed than for Small.

Applying a two-tailed t-test using a 5% confidence level and 58 degrees of freedom:

$$\frac{0.323 - 0.292}{\sqrt{\frac{.120^2 + .126^2}{30}}} = 0.98 < 2.00\,.$$

Therefore, the test conclusion is to reject H_1 in favor of H_0. That is, from these tests one can not conclude that Infopipes ECho performance latency changes from the Small to Mixed application packets.

Applying the same techniques for the hand-written communication code:

H_0: Hand-written code for Mixed and Small-QuickACK have the same average latency.

H_1: Hand-written code has a different average latency for Mixed than for Small-QuickACK.

Again, applying a two-tailed t-test using a 5% confidence level and 58 degrees of freedom:

$$\frac{0.243 - 0.194}{\sqrt{\frac{.007^2 + .00006^2}{30}}} = 38 > 2.00 .$$

In this case, the tests supports the conclusion that the two average latency values are different. Some explanation of this may arise from examining the Small packet numbers for the hand-written code in more detail.

It is instructive to delve into the small packets numbers in more detail. Initial experiments showed that the hand-written TCP "Default" case was at a clear disadvantage to all other communication software. After ad hoc experimentation with socket options, it was determined that a large portion of this delay could be accounted for by the socket's default behavior regarding TCP ACKs. By setting the TCP_QUICKACK option before each `sendmsg` call, the latency was sharply reduced. This adjustment was unsatisfactory, as it did not explain all of the performance differences to the Infopipes code despite the two source codes' apparent similarities. Further research into the

anomaly revealed the performance differences to persist despite changes to socket options, communication calls (`send` vs. `sendmsg` vs. `write`), function encapsulation, programmer, kernel version, compiler version, communication directionality (narita -> heracles vs. heracles->narita), hardware, time of day, and phase of the moon. The eventual explanation was that an open listen socket caused delays to the data in the outgoing TCP buffer and interposed a "periodic" latency length around 124 μs. During each trial, therefore, latency values would cluster around 124, 248, 352, 496, etc. Closing the open socket lowered average latency and lead to much diminished jitter.

Across types of communication software, Infopipes performance generally meets the hand-written and RPC based software packages for these two-node experiments.

SunRPC generally imposed higher latencies than Infopipes. Note, too, that the SunRPC latencies improve by changing the connection protocol from TCP to UDP and by changing the return value to void from int. TAO shows similar results to SunRPC over TCP returning int. ECho Infopipes do show a significant performance disadvantage for the small packs as compared to plain sockets. However, it is worth noting that the Infopipes abstraction allows a developer to "flip a switch" from ECho to sockets, and as long no ECho-specific features are needed, such as uploadable content filters, the Infopipe can run on the sockets layer instead. Finally, both C and C++ Infopipes over sockets proved to have competitive performance for this test. While the current discrepancy in performance between C and C++

The question arises as to why Inofpipes and hand-coded software should diverge from so much from a package such as TAO. To show from where the performance penalties arise, the applications were re-executed under the `oprofile-0.9.1` system

profiler without latency timing code. `oprofile` is a system-level profiler that uses timers to sample the CPU state at regular intervals. At each timer event, `oprofile` notes the contents of the PC, stack frame, and the identity of the running process. Naturally, functions in which an application spends more time will have more recorded interrupts by `oprofile`. Using this information, `oprofile` can statistically re-create the performance of the application and generate a report.

Initially, `oprofile` was run against executables that were dynamically linked to their communication libraries – TAO was dynamically linked to libACE and libTAO, SunRPC, Infopipes, and hand-written code were all dynamically linked into libc. However, using dynamic linking meant that function calls between libraries typically went through a jump table, the `.plt` or procedure linkage table. While this was instructive to know that the TAO tests spent about 6% of their time in the jump table switching between `libACE`, `libTAO`, and the program, it also obscured the true callgraph of the program.

By linking programs statically, `oprofile` can build a correct call graph without `.plt` interference and show the proper time-in-function statistics. Below, are the top results returned by `oprofile` – the functions in which the applications spend the bulk of their time.

Table 5. `oprofile` results for Hand-written QuickACK case, Small Packets

Hand-written code			
Client		Server	
Function	% of time	Function	% of time
sendmsg	80.0	recvmsg	94.7
smallPacketCalls	10.0		

Table 6. `oprofile` results for Infopipes C, TCP Small Packets

Infopipes C-TCP code			
Source		Sink	
Function	% of time	Function	% of time
sendmsg	65.9	recvmsg	86
do_lookup_x	4.5	__pthread_enable_asynccancel	6
infopipe_testout_port_push	4.5	__pthread_disable_asynccancel	4
_IO_setb	2.3	fillin_rpath	2
__pthread_internal_tsd_address	2.3	infopipe_testin_port_receiveloop	2
__pthread_lock	2.3		
_dl_cache_libcmp	2.3		
_dl_map_object_deps	2.3		

Table 7. `oprofile` results for Infopipes C++, TCP Small Packets

Infopipes C++-TCP code			
Source		Sink	
Function	% of time	Function	% of time
sendmsg	79.1	recvmsg	78.7
Outport_testout_port::push()	2.3	Inport_testin_port::readnet()	6.6
_dl_relocate_object	2.3	Middle_sink::middle()	3.3
Pipe_source::run_test()	1.2	_IO_new_file_xsputn	3.3
_IO_file_stat	1.2	Inport_testin_port::serviceConnection()	1.6
__basic_file<char>::close()	1.2	__gnu_cxx::__exchange_and_add(int	1.6
_dl_load_cache_lookup	1.2	__libc_init_secure	1.6
_dl_lookup_symbol_x	1.2	global	1.6
_dl_map_object_from_fd	1.2	write	1.6
_dl_mcount_wrapper_check	1.2		
do_lookup_x	1.2		
malloc	1.2		

Table 8. `oprofile` results for TAO Small Packets, Source/Client

Source	% of time
pthread_mutex_unlock	6.6
pthread_mutex_lock	6.2
memcpy	2.9
pthread_sigmask	1.9
ACE_TP_Reactor::get_socket_event_info(ACE_EH_Dispatch_Info&)	1.8
___newselect_nocancel	1.6
ACE_Select_Reactor_T<ACE_Reactor_Token_T<ACE_Token>>::wait_for_multiple_events(ACE_Select_Reactor_Handle_Set&,	1.5
TAO::Synch_Twoway_Invocation::remote_twoway(ACE_Time_Value*)	1.2
writev	1.1
ACE_Select_Reactor_T<ACE_Reactor_Token_T<ACE_Token>>::suspend_i(int)	1.1
ACE_TP_Reactor::handle_events(ACE_Time_Value*)	1.1
TAO_Transport::handle_input(TAO_Resume_Handle&,	1.1
ACE_Select_Reactor_Handler_Repository::find(int,	1.0
ACE_Lock_Adapter<ACE_Thread_Mutex>::acquire()	1.0
TAO::Remote_Invocation::init_target_spec(TAO_Target_Specification&)	1.0
ACE_Handle_Set::sync(int)	0.9
ACE_Hash_Map_Manager_Ex<unsigned	0.9
memset	0.9
ACE_Select_Reactor_T<ACE_Reactor_Token_T<ACE_Token>>::resume_i(int)	0.9
TAO_Leader_Follower::wait_for_event(TAO_LF_Event*,	0.9
ACE_TP_Reactor::clear_dispatch_mask(int,	0.9
ACE_TP_Reactor::handle_socket_events(int&,	0.9
ACE_OS::mutex_lock(pthread_mutex_t*)	0.8
TAO_Transport::drain_queue_i()	0.8
ACE_TP_Reactor::get_event_for_dispatching(ACE_Time_Value*)	0.8
_int_malloc	0.8
__read_nocancel	0.8
TAO_Wait_On_Leader_Follower::wait(ACE_Time_Value*,	0.7
throughput_small::ThrSmall::throughput_small_pckt_call(int)	0.7
TAO_Bind_Dispatcher_Guard::~TAO_Bind_Dispatcher_Guard()	0.7
ACE_Handle_Set_Iterator::operator()()	0.7

Table 9. `oprofile` results for TAO Small Packets, Sink/Server

Sink	% of time
pthread_mutex_unlock	5.1
pthread_mutex_lock	3.8
memcpy	2.6
pthread_sigmask	2.0
ACE_Select_Reactor_T<ACE_Reactor_Token_T<ACE_Token>>::wait_for_multiple_events(ACE_Select_Reactor_Handle_Set&,	1.8
ACE_Handle_Set::sync(int)	1.4
TAO_Transport::handle_input(TAO_Resume_Handle&,	1.3
writev	1.3
TAO_Object_Adapter::dispatch(TAO::ObjectKey&,	1.3
ACE_TP_Reactor::handle_events(ACE_Time_Value*)	1.2
TAO_GIOP_Message_Base::process_request_message(TAO_Transport*,	1.2
___newselect_nocancel	1.2
ACE_TP_Reactor::get_socket_event_info(ACE_EH_Dispatch_Info&)	1.2
ACE_TP_Reactor::handle_socket_events(int&,	1.2
TAO_Connection_Handler::handle_input_internal(int,	1.1
TAO::Upcall_Wrapper::upcall(TAO_ServerRequest&,	1.1
memset	1.1
strncmp	1.1
TAO_IIOP_Transport::send(iovec*,	1.1
TAO_Transport::send_reply_message_i(ACE_Message_Block	1.0
TAO_Transport::drain_queue_helper(int&,	1.0
TAO_GIOP_Message_Base::process_request(TAO_Transport*,	0.9
TAO_Adapter_Registry::dispatch(TAO::ObjectKey&,	0.9
ACE_Handle_Set_Iterator::operator()()	0.9
ACE_Select_Reactor_T<ACE_Reactor_Token_T<ACE_Token>>::suspend_i(int)	0.9
TAO_GIOP_Message_Generator_Parser_12::parse_request_header(TAO_ServerRequest&)	0.9
ACE_InputCDR::read_4(unsigned	0.9
ACE_TP_Reactor::clear_dispatch_mask(int,	0.8
TAO_GIOP_Message_Base::generate_reply_header(TAO_OutputCDR&,	0.8
TAO_Unbounded_Sequence<unsigned	0.7
ACE_OS::mutex_lock(pthread_mutex_t*)	0.7

It is easily ascertained that TAO's higher latency is a product of its library complexity and from synchronization. It is not clear exactly why TAO is spending time in synchronization code. TAO's complex object oriented design, with many small functions, is here working to the detriment of performance. Obviously, none of these functions imposes a high-penalty on its own, but in the aggregate a great deal of time is spent on tasks non-essential to the simple data transfer of the benchmark. By using code generation to avoid complex libraries, the Infopipes implementation also avoids the resultant performance penalty.

Derived from the observed latency values, is jitter, and it is another important information flow characteristic.

3.8.6.2. Jitter

The jitter metric captures the variation in latency from one application packet to the next. As stated before, it is controllable in some cases by deliberately increasing latency. Of course, the lower the jitter in a system to begin with, the more easily it is controlled.

This jitter benchmark was also for an information flow across two nodes. This was done, again, at all application data packet sizes and for all of the communication software platforms. The results are shown in Table 10 and Figure 17. Jitter was computed by computing the statistical standard deviation over the latencies recorded during each trial. For example, for the Small packet case the standard deviation was calculated for the 10,000 application packet send events. Each reported "Average Jitter" is the numerical average of the jitter reported from each trial. The reported standard deviation is the variation amongst these trials.

Table 10. Results of synthetic jitter benchmarks.

Communication Software				Trials	Avg. Jitter (ms)	Std. Dev. of Jitter (ms)
Small	Hand	C	Default	30	0.010	0.001
			QuickACK	30	0.028	0.005
			CloseListen			
	Infopipe	C	ECho	30	0.046	0.038
			TCP	30	0.013	0.016
		C++	TCP	30	0.003	0.002
	SunRPC	int	TCP	30	0.059	0.036
		void	TCP	30	0.034	0.038
		int	UDP	30	0.078	0.029
		void	UDP	30	0.054	0.044
	TAO		twoway	30	0.025	0.009
Mixed	Hand	C	TCP	30	0.061	0.001
	Infopipe	C	ECho	30	0.050	0.050
			TCP	30	0.013	0.016
		C++	TCP	30	0.014	0.015
	SunRPC	TCP	int	30	0.112	0.047
		UDP	int	30	0.055	0.015
	TAO	C++	twoway	30	0.060	0.004
Large	Hand	TCP		30	0.062	0.001
	Infopipe	C	ECho	30	0.044	0.024
			TCP	30	0.030	0.017
		C++	TCP	30	0.038	0.022
	SunRPC	TCP	int	30	0.035	0.032
			void	30	0.030	0.026
	TAO	C++	twoway	30	0.040	0.005

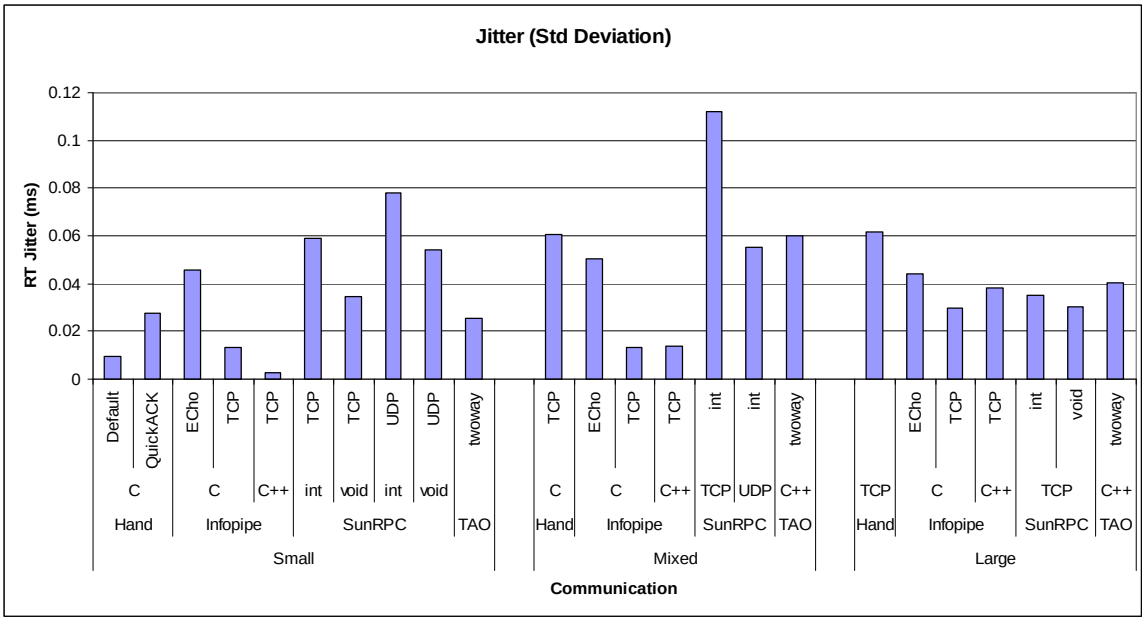


Figure 17. Synthetic benchmark jitter for two nodes

As evident from the graph, jitter exhibits very little in terms of trends across the three types of applications packets. However, that latency for Infopipes is low compared to the RPC protocols.

3.8.6.3. Throughput

Throughput is most important for applications streaming large amounts of data on a continuous basis. Each throughput benchmark trial is measured by first taking a time stamp, then sending a series of application packets (10,000 for Small; 1,000 for Mixed and Large), and finally taking final timestamp. The overall throughput can be calculated from this by dividing the number of application bytes sent, as the benchmark measures channel capacity through the communication software, by the time of the test. The results of 30 trials for each benchmark are summarized in Table 11 and Figure 18.

Table 11. Results of synthetic throughput benchmarks.

Communication Software				Trials	Avg. Throughput (KBps)	Std. Dev. of Throughput (KBps)
Small	Hand	TCP		30	3171.70	20.85
	Infopipe	C	ECho	30	228.62	1.31
		C	TCP	30	2244.52	65.49
		C++	TCP	30	2302.65	52.91
	SunRPC	TCP	int	30	22.72	9.61
			void	30	19.59	9.98
		UDP	int	30	20.39	6.51
			void	30	26.08	7.45
	TAO		twoway	30	16.69	2.64
Mixed	Hand	TCP		30	52998.40	1600.67
	Infopipe	C	ECho	30	8157.35	74.24
		C	TCP	30	52262.07	1996.72
		C++	TCP	30	53215.95	2521.70
	SunRPC	TCP	int	30	395.21	74.59
		UDP	int	30	758.57	20.82
	TAO		twoway	30	518.33	36.51
Large	Hand	TCP		30	101551.30	220.30
	Infopipe	C	ECho	30	94696.92	10004.13
		C	TCP	30	105099.13	3464.46
		C++	TCP	30	105443.38	2481.49
	SunRPC	TCP	int	30	25291.74	1647.74
			void	30	25120.80	1261.85
	TAO		twoway	30	31352.90	222.56

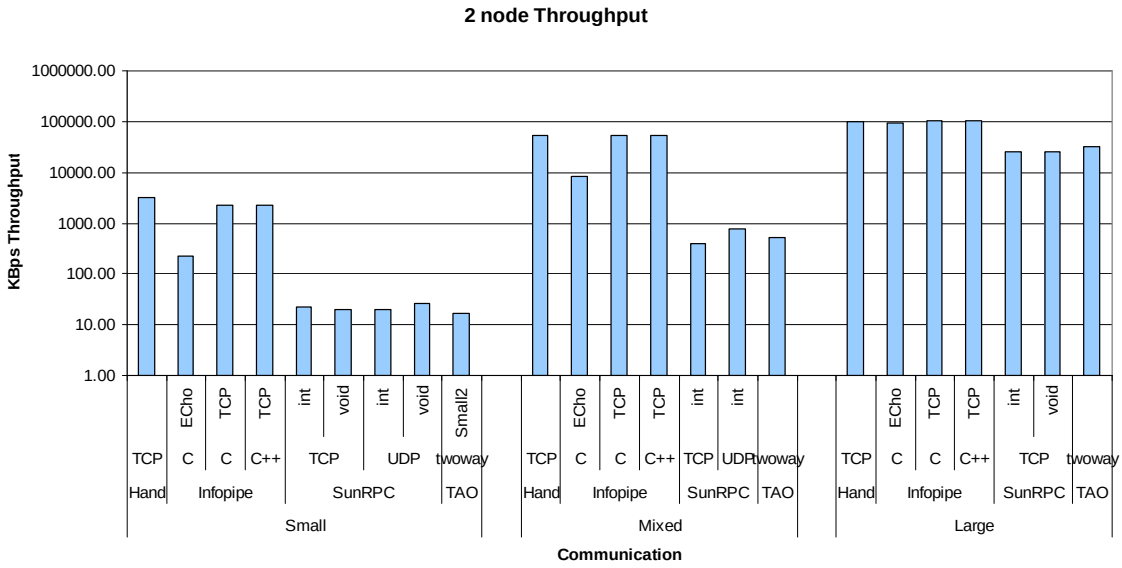


Figure 18. Synthetic benchmark throughput for two nodes (logarithmic scale).

The data indicate that the request response protocol of SunRPC and TAO serve to limit the effective bandwidth achieved by those two communication software packages. Both the hand-written code and the Infopipes code can both make effective use of the TCP connection. This is because these communication layers do not demand a reply from the application level recipient at the sink end of the data connection. Note, too, that Infopipes-ECho also provides better performance for throughput than SunRPC or TAO despite the fact that its latency was comparable for each application packet type.

3.8.7. Three Node Synthetic Benchmarks

One of the hallmarks of information flow applications is they often include multiple nodes of computation which data must traverse en route to its ultimate sink or node where it will be useful to a consumer. The three node benchmarks are designed to provide insight into the effect that communication software decisions may have in the key performance areas of jitter, latency, and throughput.

3.8.7.1. Latency

For three node configurations, latency remains a significant concern to the application developer. The benchmark approximates the end-to-end latency in Infopipes by sending the data to the middle node, immediately forwarding it to the sink node, and returning a reply, via socket, from the sink node back to the source node. In the RPC-type protocols, the function call to the sink node is done from within the remote function of the middle node.

The results for the three node synthetic benchmark are given in Table 12 and Figure 19. These results largely echo the results for the two node latency benchmarks. As far as general trends in the graph, the three node latency measurements largely mirror their two node latency counterparts.

On the other hand, if one compares latency relative to the two node metrics, then there is a significant slowdown seen in the RPC cases as shown in Figure 20. Each of the SunRPC TAO twoway based benchmarks required to pieces of return data: first from the sink to the middle node, and second, from the middle node to the source. This significantly increases the end-to-end latency penalty in this benchmark. Given the semantics of TCP, as a reliable protocol, there is no reliability gained by returning the information. While the penalty factor does fall off as the data packet sizes increase, it still has significant impact for TAO even in the large application packet case.

Another way of looking at the metrics from the threenode case is to compare the results on a relative basis to the two node case. Intuitively, the communication time should be about twice as long for the three node case as for the two node case because there are twice as many communication links to traverse. However, in the RPC-style benchmarks (TAO twoway and SunRPC), as shown in Figure 20, the performance

penalty is closer to three times the corresponding two node benchmark. In such cases, the extra overhead can be attributed to the need to wait for remote functions to return.

This particular benchmark points to a potential pitfall in RPC programming – the semantics governing RPC imply that remote calls may become dependent on arbitrarily long chains of remote calls when information is processed in a pipelined fashion. Latency might then become arbitrarily large.

Of course, if a developer for an application knew that the RPC call would result in a second remote call, he could construct calls using non-blocking semantics. Likewise, if a service developer engaged in deploying an RPC function knew that it would accessed by potentially. It may be a fair assumption to make that if the programmer is aware the remote call is designed for and will be deployed in an information flow application, then the programmer can use threading to decouple packet processing of the first stage from later stages. On the other hand, such complex programming may be overkill in some applications, but not others.

It is important to distinguish this type of middleware performance concern from an application-level performance concern. After all, similar delays could just as well arise from long-running computation within a function in the information flow. That is, a particular stage of the information flow application may be compute bound and limit the processing rate and impose high latencies. As noted though, this becomes an application performance problem and is, to a degree, independent of the middleware whereas the latency introduced in this benchmark is a direct result of the middleware choice.

Table 12. Three node synthetic latency benchmark

Communication Software				Trials	Latency (ms)	Relative to Two Node
Small	Hand	C	TCP	30	0.454	0.91
			TCP-no Nagle	30	0.440	1.81
	Infopipe	C	TCP	30	0.316	1.80
		C++	TCP	30	0.364	2.11
	SunRPC	int	TCP	30	0.838	3.25
			UDP	30	0.754	3.49
	TAO	C++	twoway	30	0.879	3.43
Mixed	Hand	C	TCP	30	0.496	2.55
	Infopipe	C	TCP	30	0.377	2.33
		C++	TCP	30	0.384	2.88
	SunRPC	int	TCP	30	0.794	2.26
			UDP	30	0.795	3.87
	TAO	C++	twoway	30	0.888	2.84
Large	Hand	C	TCP	30	0.602	1.85
	Infopipe	C	TCP	30	0.613	2.05
		C++	TCP	30	0.616	2.31
	SunRPC	int	TCP	30	1.013	2.07
	TAO	C++	twoway	30	1.166	2.93

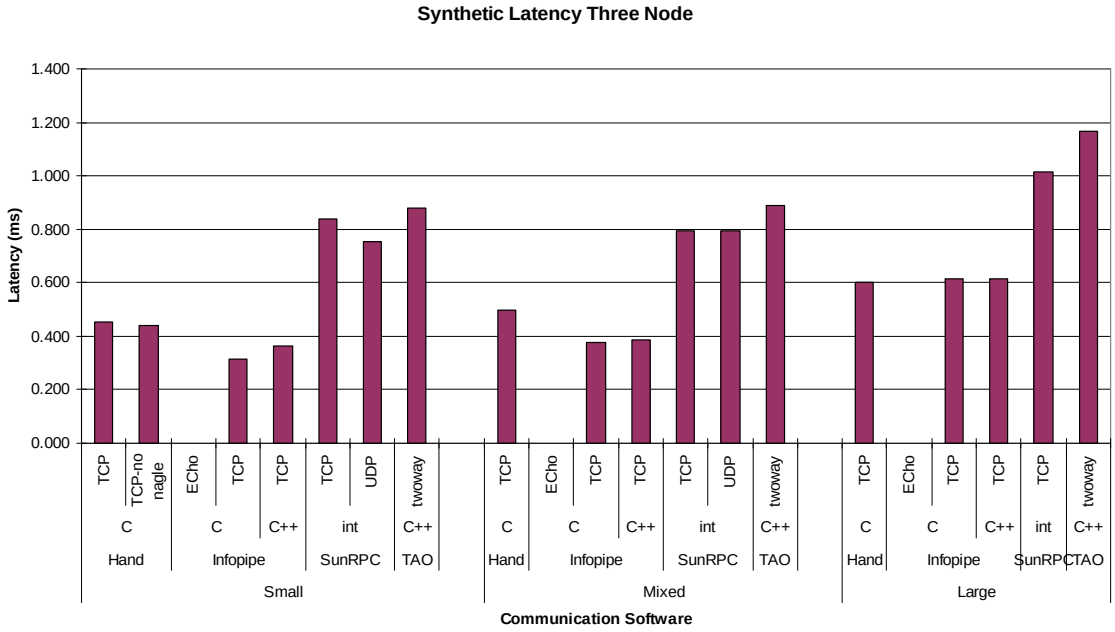


Figure 19. Synthetic benchmark latency for three nodes.

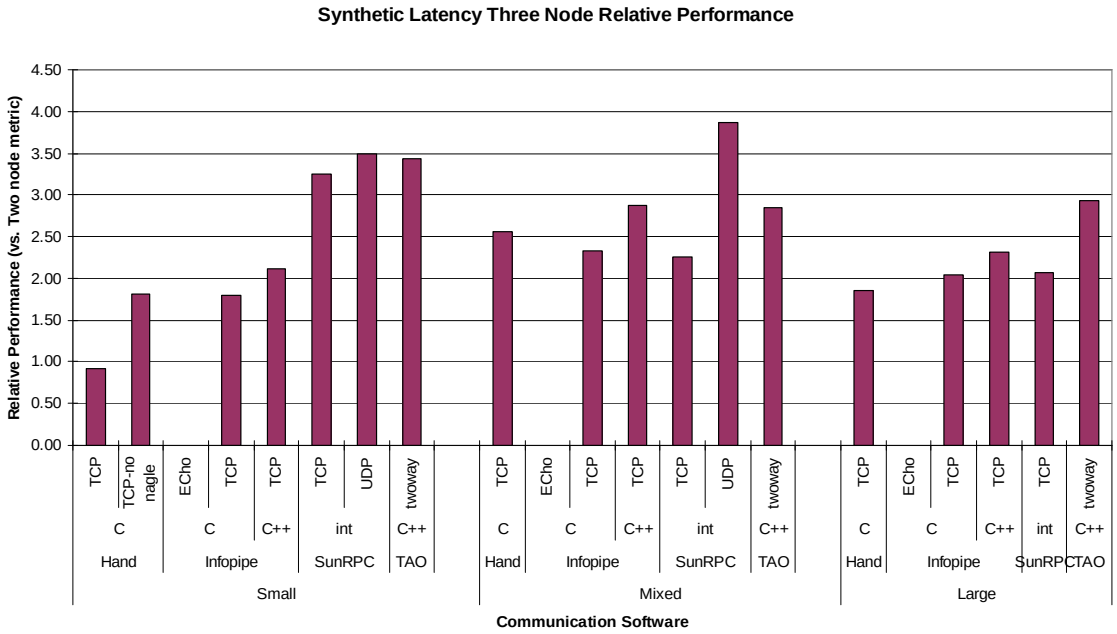


Figure 20. Synthetic latency benchmark for three node case relative to two node case.

3.8.7.2. Jitter

Jitter for three node benchmarks are presented Table 13 and Figure 21. As in the two node version of the benchmark, jitter is highest for the SunRPC case. TAO jitter is lower than SunRPC because it employs synchronization which effectively “slots” latencies into predictable times.

Table 13. Three node synthetic jitter benchmark

Communication Software				Trials	Jitter (ms)
Small	Hand	C	TCP	30	0.120
			TCP-no Nagle	30	0.086
	Infopipe	C	TCP	30	0.085
		C++	TCP	30	0.114
	SunRPC	int	TCP	30	0.212
			UDP	30	0.303
	TAO	C++	twoway	30	0.188
Mixed	Hand	C	TCP	30	0.038
	Infopipe	C	TCP	30	0.132
		C++	TCP	30	0.127
	SunRPC	int	TCP	30	0.257
			UDP	30	0.271
	TAO	C++	twoway	30	0.167
Large	Hand	C	TCP	30	0.034
	Infopipe	C	TCP	30	0.032
		C++	TCP	30	0.035
	SunRPC	int	TCP	30	0.054
	TAO	C++	twoway	30	0.221

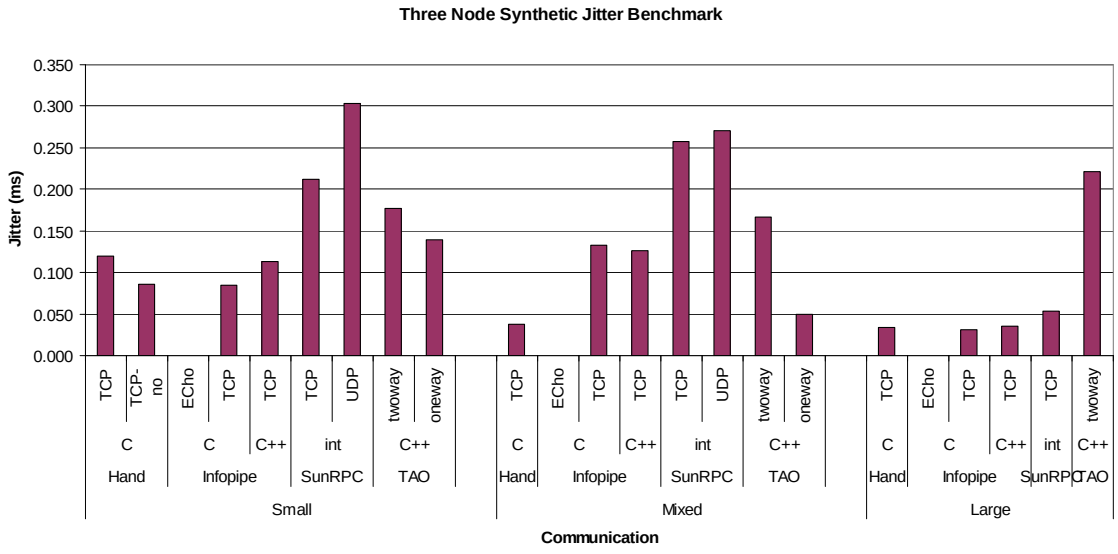


Figure 21. Synthetic jitter benchmark for three node case.

3.8.7.3. Throughput

In a three node case, throughput is measured as the number of bytes passing between processing nodes in an information flow. In this case, bytes map directly to application data packet transfers. This provides for comparability between different data types used in the throughput experiments. Effectively throughput measures the number of bytes the source node can transmit to the sink node in a given time period.

Throughput for the three node synthetic benchmark shows that RPC calling semantics impose serious overhead as compared to information flow-centric or hand-written approaches to the problem. In the table and the figure, throughput for the three node case is even more severely curtailed than in the two node synthetic throughput benchmark.

On a relative basis, too, hand-written and generated Infopipes code performs as expected with about half the throughput of the two node case. In these experiments, each node was a single-homed device, so that incoming and outgoing traffic at the center node was sharing the same network device.

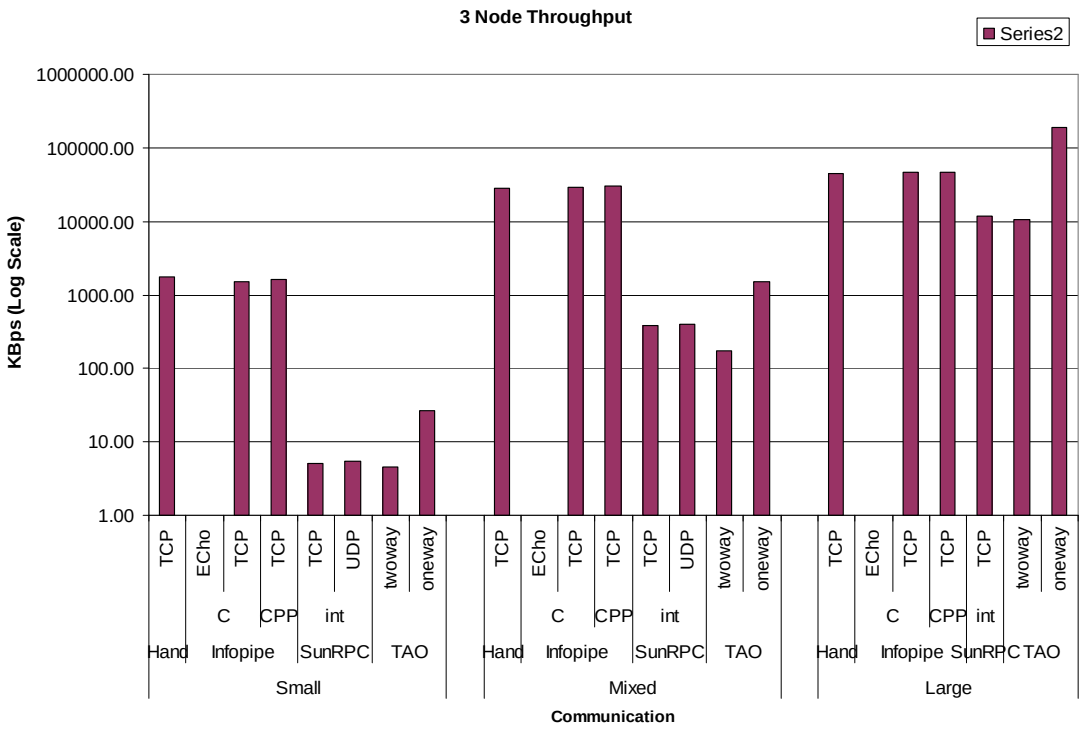


Figure 22. Synthetic throughput benchmark for three node case.

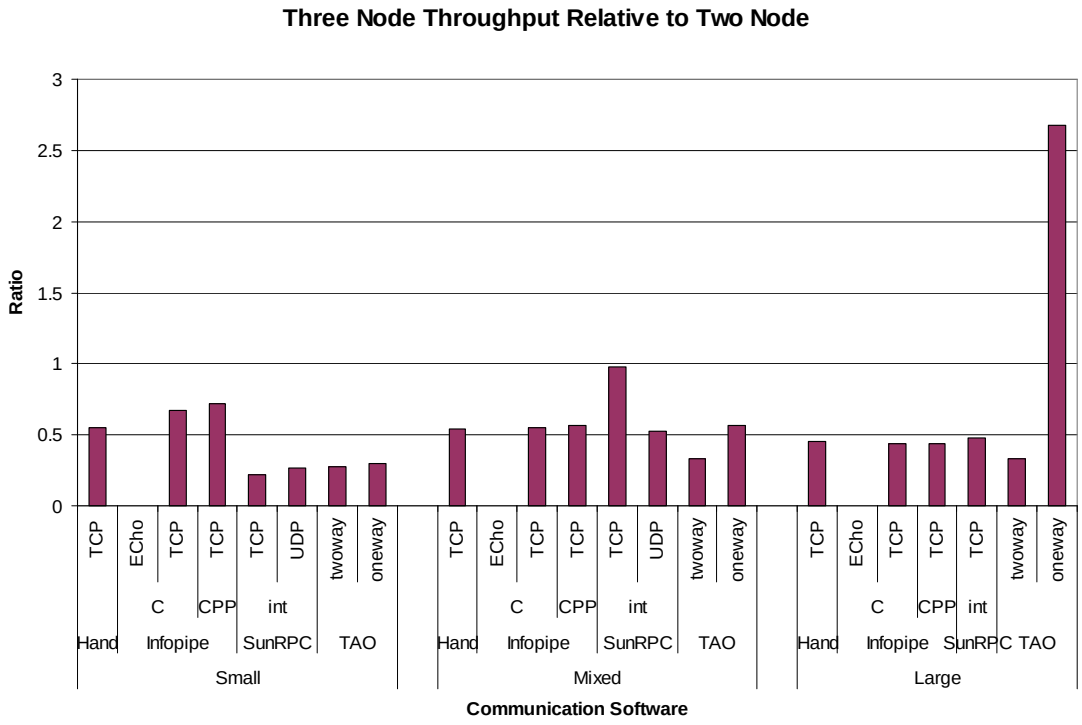


Figure 23. Synthetic throughput benchmark for three node case relative to two node case.

3.8.8. Application Based Benchmarks

When characterizing system software and tools, synthetic benchmarks may go a long way towards revealing the underlying factors determining system performance, but may not, in fact, reflect the true performance of the system when given an application workload. Therefore, it is also helpful to establish a system benchmark based upon an application scenario which reflects a simplified but plausible use case for the software in question. For the ISG, the test was again against generated communication software packages, those being the Infopipes, SunRPC, and TAO, using an application based on the Multi-UAV scenario.

This benchmark is also executed in two different variants. First, the two node application based benchmark consists of the transfer of images between a source node and a sink node. The three node benchmark, however, is not the straightforward transfer of data between the source, a middle node, and sink. Instead, a transformation step is added in which the image is transformed from a full color, 24-bits per pixel representation into a grayscaled image. This transformation involves operating and transforming the memory occupying the data for transfer. By doing so, the benchmark characterizes an application and the effect of the communication software under conditions where significant processing must be carried out on the data.

As before, the benchmarks consider two node and three node benchmarks for each of latency, jitter, and throughput.

3.8.8.1. Latency

For Figure 24, dark colors plot the performance of the communication software for the two node case. For this case, the Infopipes/TCP versions perform slightly better

than the SunRPC and TAO versions. The Infopipes/ECho version performs on par with the SunRPC version.

For the three node case, there is a significant degradation in performance for the SunRPC and TAO software packages as compared to the Infopipes versions. This is due to extra data copies within the communication stack. In the Infopipes case, the image can be transformed in the middle node by directly placing the computed output grayscale image in the output port data structure. From there it is sent using scatter/gather based system calls to the sink.

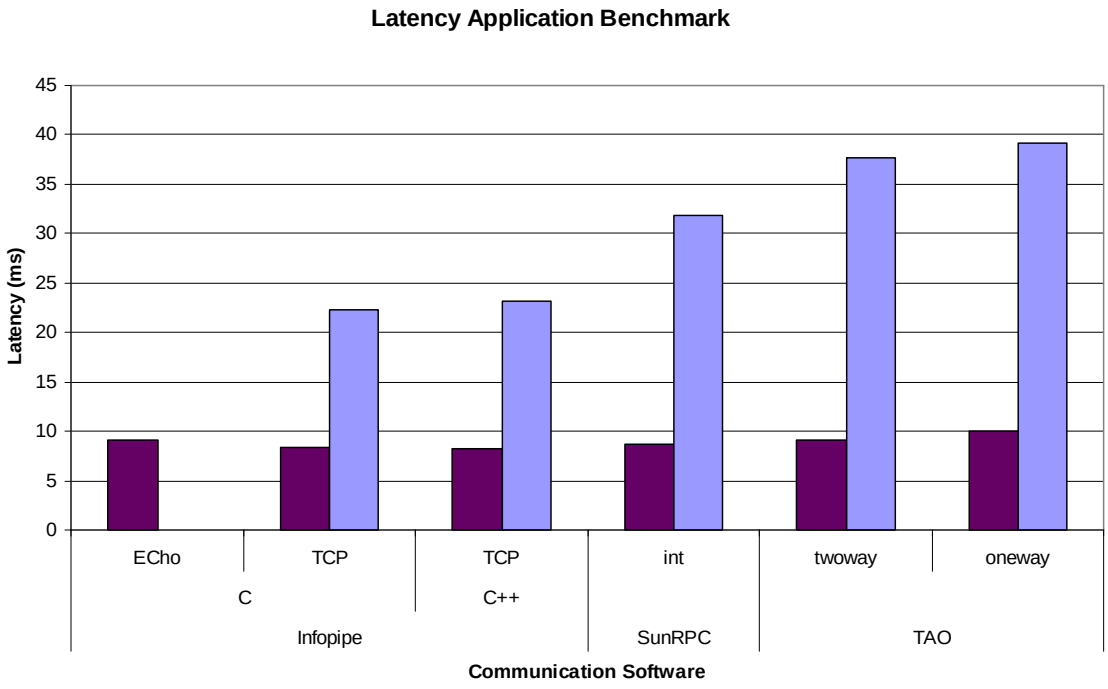


Figure 24. Application latency benchmark.

3.8.8.2. Jitter

For jitter, SunRPC returns the most jitter in the middleware and TAO the least. While TAO does have lower jitter, due to synchronization, this comes at the cost higher latencies – particularly in the three node application benchmark.

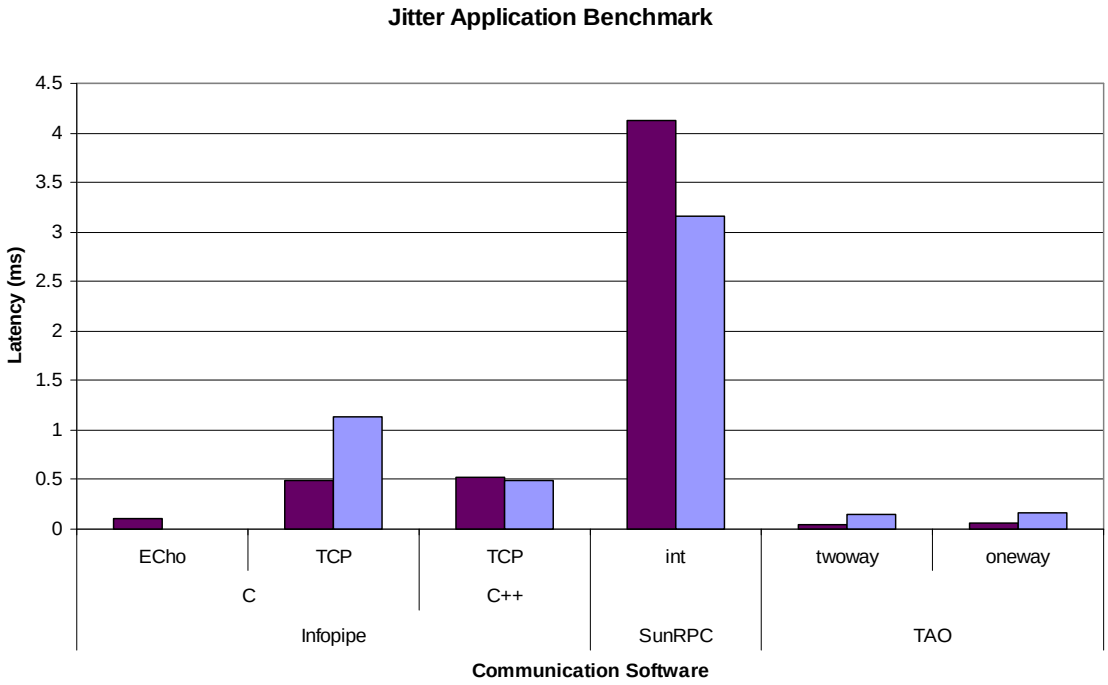


Figure 25. Application jitter benchmark.

3.8.8.3. Throughput

In the application throughput benchmark, Infopipes again provide good throughput as compared to other communication software packages. Particularly, interesting, however, is that the even when there is a large amount of computation work, as in the three node case, Infopipes still perform better than the SunRPC and TAO twoway calls. This evidence points to the conclusion that the additional overhead imposed by the request/respond semantics are non-trivial and do not “go away” in applications in which performance may be dominated by some other factor than communication (in this case computation in the middle node).

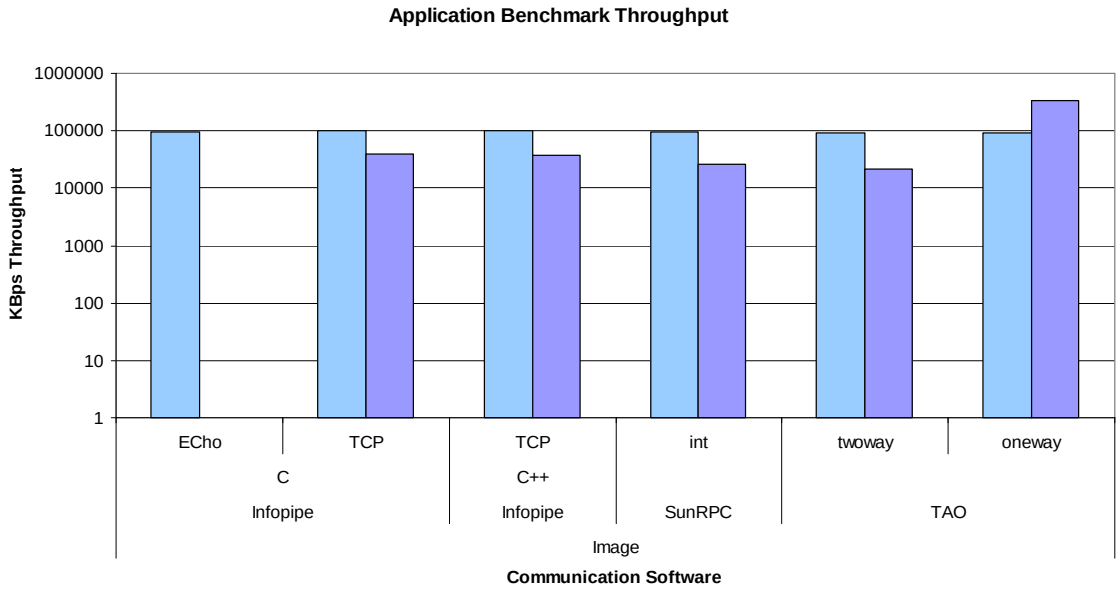


Figure 26. Application throughput benchmark.

3.8.9. Benchmarks Conclusion

In comparing Infopipes to other communication software, benchmarks indicate that Infopipes typically perform at a level comparable to existing types of communication software. Furthermore, the benchmarks indicate that the request/respond semantics and control inherent to RPC-style communication software may not be appropriate for many types of information flow applications as it imposes unnecessary coordination over head between stages in the application pipeline. Interestingly, this overhead appears to be persistent up to even large data transfer sizes and even when computational load from the application itself is non-trivial.

Given the Infopipes apparently perform comparably to existing middleware in synthetic benchmarks, the question turns to their performance in application settings. There are two applications that demonstrate the utility of ISG-generated code as it might be used for real-world applications.

3.9. Application 1: Distributed Linear Road

3.9.1. Scenario

The Linear Road benchmark has been developed by researchers to test new stream query systems [5]. It is based on a dynamic toll scenario under consideration by highway agencies in several states to help combat congestion and encourage drivers, through higher tolls, to travel at off-peak times. In the benchmark, there are a series of multi-lane highways, and each highway is divided into segments. Cars on the highway provide regular position and velocity updates which are tracked by the query system. Then as a car approaches a highway segment, it is notified of its toll calculated based on observed traffic volume. In the real world, a driver can opts to accept the toll and continue on the highway or exit. If the driver accepts the toll, then the toll system debits an account. Ultimately, the benchmark's measure is how many "highways" a query system can support before responses a returned too slowly – i.e., response latency is violated.

It is easy to see that QoS is a natural and important requirement for Linear Road: the toll calculator must execute efficiently and quickly to inform drivers in time for them to make safe decisions about whether to continue on the toll road or, if the toll is too high, then exit at the next segment. Consequently, latency becomes an important end-to-end property that must be monitored closely – each position report must be answered within a few seconds with updated toll information. Once latencies become too long, drivers may attempt unsafe exits, or the toll authority may be forced into a default toll policy at the expense of lost revenue opportunities.

The complete Linear Road package has a set of Data Generator processes that create a stream of location data to simulate car locations on the toll road. Along with

location data, the Linear Road also creates a set of queries over the data it produces, including Position Reports, Account Balances and Daily Expenditures, and Travel Time Estimation. For a full benchmark, the system must execute static queries and continual queries simultaneously. The generated data is fed through Data Drivers to the query system. By default, Linear Road produces location data that correspond to a location report for every 30 seconds of "real" time that passes in the simulation.

To execute the Linear Road queries, the application relies on STREAM [4]. Running it requires a continual query (CQ) script (see Figure 27) and a second file of system configuration parameters. The CQ script contains three types of information. First, it contains a description of the input tuples as an association of field names and types. Second, it contains a file path location from which STREAM can read tuples. Lastly, the script contains continual queries described in the Continual Query Language (CQL) [CQL]. CQL manipulates streams (a time-stamped and time-ordered sequence of tuples) and relations (time-varying sets of tuples) through various operators [6].

```
table : register stream CommonInputStr
         (Vid integer, Speed integer,
          XWay integer, Lane integer,
          Dir integer, Seg integer);
source : /hc283/stream_proj/source/db0

vquery : select Vid, XWay*1000+Dir*100+Seg,
           Speed from CommonInputStr;
vtable : register stream SegSpdStr
           (Vid integer, Seg integer,
            Speed integer);
...
query : select Vid, Seg, Lav,
          Toll from OutStr;
dest : /hc283/stream_proj/out/lrtest0
```

Figure 27. Excerpt of a STREAM script.

STREAM assumes the input tuples arrive in time-stamp order. Furthermore, STREAM is undergoing rapid development, and for this experiment a version was not available that implemented persistent storage. Therefore, the benchmark is distilled to the essential continual query: calculating variable toll amounts on a volume basis. In this query, latency is incurred in internal buffers that STREAM automatically creates as part of the query plan. For this paper, an Infopipe feeds STREAM tuples from vehicles on a highway in a serialized fashion from a generated simulation trace and then measure the latency as they are collected as illustrated in Figure 28.

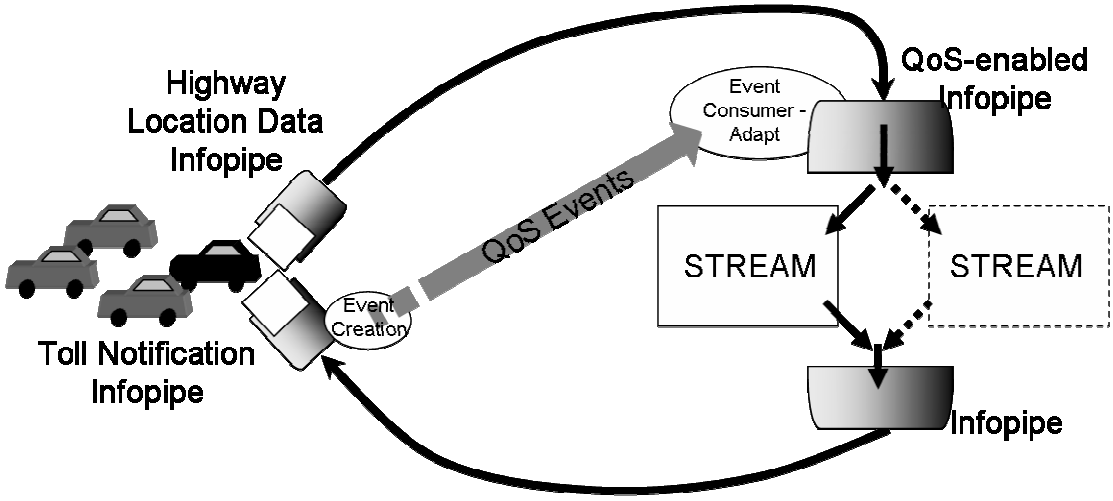


Figure 28. In the Linear Road benchmark, cars transmit location data to the STREAM CQ server. QoS feedback events flow from the cars to the Infopipes wrapping the STREAM servers.

As mentioned earlier, STREAM also supports neither distributed computation nor QoS – both of which are inherent to the Linear Road scenario. the next section describes how add these features.

3.9.2. Implementation

The first task was to distribute Linear Road by wrapping each of the three application units (data source, STREAM server, and data sink) in an Infopipe. Currently, the experiments, as did a prior Linear Road benchmarking project, use a single data stream of highway information rather than creating a system of thousands of cars, each generating its own data. Instead, these experiments the aggregated highway data flows through an Infopipe to a remote machine running the STREAM system, which then feeds the data out after query execution to an application sink, again via Infopipe. The STREAM server is wrapped with one input-only and one output-only Infopipe, streamReceiver (receiving from the car data source) and streamSender (sending back toll information to the cars), respectively. STREAM itself is hardcoded to read and write data only from files specified in the CQL. By embedding Unix pipe-creation code in the wrapping Infopipes, STREAM could be connected to other data sources. The Infopipes fed information from network sources through these pipes.

An event-based model for controlling QoS is a natural fit since, generally, QoS is only relevant when some system event triggers quality evaluation. The QoS co-system is readily capture as aspects to the Infopipes application, and use the AXpect weaver to integrate this code into the generation and deployment phase of the application. This achieves two goals. First, it allows logical, high-level separation of the QoS system. This type of abstraction enhances understandability. I.e., the developer can first create and debug a non-QoS system that has no QoS code to complicate the testing or performance profiling of the raw application. Second, abstracting the QoS system into a separate specification as standalone code offers the opportunity of reuse in later information flow systems. Furthermore, this method of packaging allows the creation of a distributed QoS

service which, in operation, is run distributed as three smaller pieces. In fact, a great deal of the QoS code applied to this problem was originally generated for the earlier image streaming application.

If Linear Road runs with no QoS support, then it quickly violates the latency policy – the response with the toll amount is not returned quickly enough after a car sends its location. However, the latency is introduced primarily by the query engine and its buffers while the CPU remains relatively free. To detect this, module on the data sinks returns latency measurements to the source Infopipes according to definitions within the WSLA. The receipt of a toll tuple at the DataSink that is late triggers a WSLA "Notification" event to be returned via the feedback channel.

If observed latency goes out of range, the monitor triggers an adaptation into a low-latency mode of operation. It does this by spawning a second copy of the query system, i.e., it spawns another STREAM instance. Based on highway numbers in the tuples, it splits the incoming tuple stream in two, and farms tuples from half the highways to the second server. This lowers latency because STREAM can use the second processor installed on the computer. Using two servers means each query engine is handling fewer tuples; they can serve all tuples more quickly to reduce latency.

The AXpect weaver installs the QoS implementation on the base code automatically during code generation. The implementation is as several small aspects, as it is easier to develop, deploy, and debug the QoS behavior by working with relatively small pieces of source code. The aspects were developed as follows:

The first aspect measures latency of data streaming through the network. Since Linear Road tuples are generated with time stamps, the aspect need only observe the time stamps at the destination end of the application and calculate elapsed time.

Next, an aspect introduces a feedback channel that returns timestamp data from the data sinks to the QoS monitor. A feedback event is generated every time that a data sink receives a data from the query server. For future flexibility, feedback information relays through both Infopipes connected to the query system. The feedback channel and event handler was reused, in fact, from an earlier project.

Next, on top of the control channel, an aspect reads and implements the QoS specification from the WSLA and converts it into source code.

New code is inserted inside the SLA code that implements replication of the query server. In doing this, the aspect must create two system structures. First, a new Unix pipe must be created with a `mkfifo` call. Second, it emits new CQ script file that has been parameterized with the new FIFO name of the Unix pipe as the tuple source. For writing output tuples, the new query server can multiplex over the same FIFO as the original query server.

Now that replicated STREAM-server aspects exist, the system needs code, also in an aspect, that distributes the tuples between the two query server copies. It does this by examining highway numbers and distributing half the highways to the replicated service, and half to the original service.

Finally, the system must avoid transient pricing errors. Since the new query server has no traffic information, it must be fed information for some time before tolls generated

by the system are consistent with the original tolls. This aspect imposes a 60-second delay after starting to new STREAM server before accepting results from it.

3.9.3. Evaluation

The evaluation was conducted on three PIII-800 dual processor machines. There was one machine for each of the system components (DataSource, QueryServers, and DataSink), and when two STREAM servers were active, they both resided on the same physical machine. The QoS-enabled system was compared to a non-QoS-enabled system using the Linear Road benchmark for 1 to 32 highways.

The experiments capture system performance as both latency and "good" throughput. The primary metric of concern was latency since it governs the timeliness for safe driver decision. Average latency is reported for each load level. Of course, even when average latency is high, some tuples may be processed in time for safe driver decisions. To examine this, a second metric of interest is "good" throughput. Instead of counting all tuples processed as regular throughput does, good throughput only counts those that arrive at the DataSink on time.

The non-QoS system, as expected, demonstrated inability to cope with high traffic levels. In Figure 29, the non-QoS curve shows that as load increases, latency grows slowly at first, but then at about 13 or 14 highways, latency rises dramatically indicating the load has exceeded the optimal operating region for STREAM. However, adding the replication aspects and metrics monitoring aspects to the system maintains latency values in the safe region for even the higher load factors, as shown in the QoS curve.

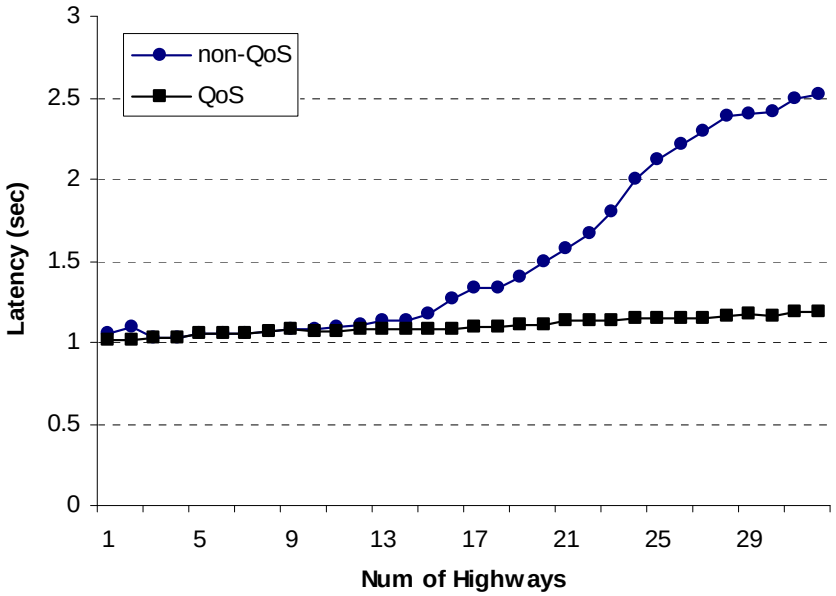


Figure 29. Average latency, non-QoS and QoS. QoS-enabling keeps latency under 1.5 seconds whereas with no QoS latency reaches 2.5 seconds.

Throughput provides an interesting perspective on the behavior of the overloaded non-QoS systems. Figure 30 provides throughput at the input and "good" throughput at the output of the STREAM Infopipes. Note that input throughput increases steadily under system load as more cars on more highways push data towards the STREAM server. On the output side, however, the overloading has the effect of reducing the absolute "good" throughput of the system. In other words, increasing the load on the system reduces the absolute amount of useful data the system is able to provide. After adding the QoS event monitor and adaptive code, the lower curve in the graph, the Linear Road application is able to provide low latency information to the requesting cars and maintain "Good" system throughput in relation to total throughput. Note, too, that for QoS-enabled case the input side total input throughput also improves slightly because neither the system nor STREAM's internal buffers become overwhelmed.

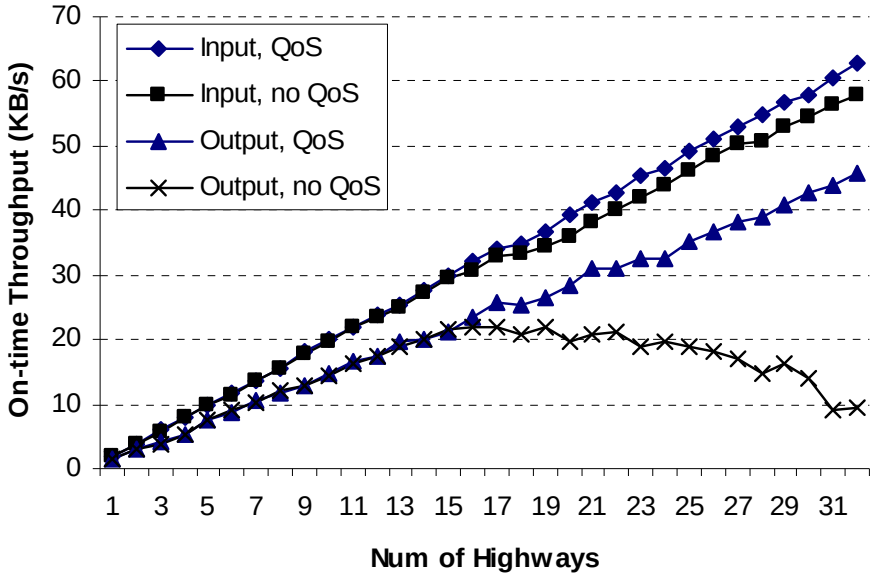


Figure 30. Throughput for input and "good" throughput for output. Higher input throughput under QoS results from more efficient servicing of buffers. Higher output throughput is from more tuples being "on time".

3.9.4. Linear Road Summary

Wrapping the query system and application components in Infopipes enabled a quality of service aware application easily and in a reusable fashion. Employing the Infopipes Stub Generator allowed reusable communication stubs; writing aspects encapsulated reusable QoS components which were woven in to the application by AXpect.

This application also illustrates how important addressing QoS can be in information flow applications. Addressing the QoS as a problem apart from application flow allowed the use event-based servicing of latency violations rather than a continual information flow.

3.10. Application 2: A MultiUAV Scenario

3.10.1. Scenario

A distributed image streaming application illustrates the functions performed by the Spi/ISG toolkit. From this, implementing a WSLA document via the AXpect weaver to impose resource constraints and add adaptation to the image source service demonstrates the utility of the AXpect weaver. In the application, two parties a sender and receiver, write a contract in which the sender generates images for the receiver. In the base implementation, the sender transmits images to the receiver at an unconstrained rate. Following this, implementing a WSLA document adds a CPU usage constraint so that the receiver measures the resource usage of the sender; returned CPU data allows the sender so the sender can adjust its sending rate and respect receiver constraints.

Operational requirements of this simple sender/receiver application are common to many distributed information flow applications. For its part, the receiver must create a socket, publish connection information, and wait for an incoming connection. The sender follows a complementary series of steps. It creates a socket, looks up the receiver's connection information, and then establishes the socket connection. In the steady state, the sender transmits data to the receiver through the connection. In the base implementation, however, it is easy for an overeager sender to swamp the receiver with too many images and thereby use up a disproportionate amount of receiver resources which may be needed for other tasks. A WSLA allows the sender and receiver to manage this situation in the experimental section.

This type of sender/receiver is plausible for multiple scenario variations. For instance, the receiver may wish to perform compute-intensive transformations on the data, or the receiver may be collecting images from multiple sources (possible even from

multiple network segments) at the same time. Or, the receiver may merely need to reserve processor capacity for local tasks. In these cases, it is useful for a rate limiter to be programmed into the sender which responds to receiver issued WSLA information messages about CPU use.

For the base scenario, there are a total of fourteen files generated each for the sender and receiver:

- `sender.{c,h}` or `receiver.{c,h}`: the datatype declarations, the middle function of the pipe, its startup, and its shutdown code
- `ppmOut.{c,h}` or `ppmIn.{c,h}`: header files for the communication functions and source files implementing marshalling, communication, and connection establishment
- `runtime.{c,h}`: header and library functions for support of connection establishment. There is one of each of these files for the sender and receiver.
- A `Makefile` for each sender and receiver.

When the application runs, it first calls the startup code for the pipe. This in turn calls the startup code of each connection for opening and connections. Once startup is complete, the pipe enters its main running phase, which consists acquiring data and submitting it to the communication layer in a continuous loop. The communication layer then manages the network transmission. Communication is asynchronous between the sender and receiver.

3.10.2. Implementation

The base scenario simply allows the sender to transmit data unchecked to the receiver using the base code generated by the templates. To add rate-limiting functionality, then, to the base implementation requires the following changes to a base sender-receiver implementation:

- Add support for resource metrics to the receiver reevaluated each time the receiver processes an application packet. Requires code added to the receiver when it initializes and when it processes each packet.
- Add a reverse channel for WSLA information messages from the receiver to the sender. This requires discovery and connection code on the client and receiver plus a mechanism to multiplex and demultiplex control messages.
- Add rate CPU metric code to the receiver which marshals and sends informational messages to the sender about the observed metric under a chosen reporting policy (e.g. windowed vs. un-windowed). It builds on the functionality of the CPU monitor aspect and the control channel. So, those aspects must be present first.
- Add rate control code to the sender. This code must retrieve messages from the control channel, demultiplex them, and behave appropriately. It again depends on the control code being applied first. The sender "throttles back" by sleeping after image transmission if the receiver reports greater than 20% CPU usage.

Figure 31 illustrates the application and aspects. Note that in addition to crosscutting the base design of the application, several aspects crosscut other aspects.

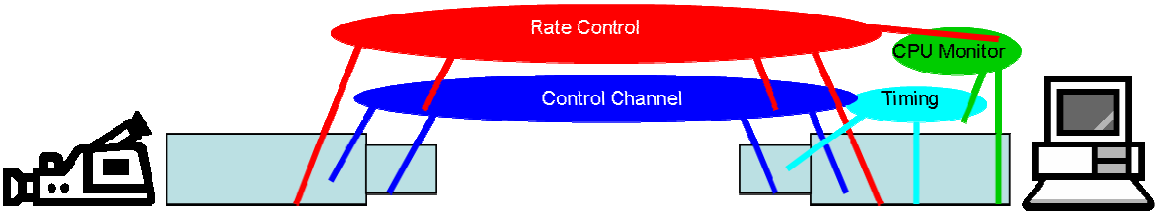


Figure 31. The QoS-aware image streaming application.

Implementing the aspects proceeded in several steps. First, since this was the first applications of aspects, it motivated the first joinpoints added to the generator templates. A total of 18 joinpoints were added to the base template code. For the most part, these additions corresponded to each major syntactic or logical unit of code

Seven aspect files encapsulated the monitoring, control, and adaptation behavior of the application. Two aspects were on the sender side: `control_sender`, which implemented the sender-side control channel, and `sla_sender`, the implementation of the sender's response to receiver rate requests. Five aspects participated in QoS on the receiver side: `timing`, which provided base timing measurements for CPU usage computation, `control_receiver`, an implementation of the receiver-side control channel, `cpumon`, which monitored CPU usage, and `sla_receiver`, which sent rate messages to the sender. Each aspect corresponded to one XSLT file.

The control channel aspect placed the bulk of functionality in files separate from code generated for the base implementation. That is, the aspect introduced new files in addition to the files generated for plain Infopipe communication. It added startup code and make rules to the sender and receiver output files. Altering the Makefile allowed automatic compilation of the extra files for the pipes, and adjusted the compile and link

flags by adding required libraries like `–lpflags` for supporting the separate thread of the control channel.

Modifications on the sender side illustrate how disruptive even relatively simple additions can be to the application. First of all, the sender must establish a control. To avoid interference of the control channel with the main communication of the application, control channel service executes in a separate thread. This means that pipe startup entails forking a new thread, creating a socket within that thread, and connecting to the receiver's control socket. Next, to support for the rate control, the startup code must also initialize rate information variables. In this case, this entails setting the sender's sleep flag and guard variable to 0 (the guard variable allows turns on and off the throttle control if the sender is allowed to send at its maximum rate). In addition to this startup complexity, the rate control aspect inserts into the control channel's demultiplexing function code that routes incoming control information to the "`rateCmdReceived()`" function, which takes proper action. Furthermore, to implement the rate throttling, the rate control aspect inserts a guard statement and `usleep()` call after the Infopipe completes its data transmission. Each of these changes involves installing variables at various scopes (global, module, and local) and in multiple header files. Finally, since new files added to the application required insertion of the aforementioned `makefile` rule and add the corresponding object files and flags to the link list.

The timing aspect hooks on to all join points that designate an executable block of code. This can be done in an efficient fashion by using the pattern matching to select entire sets of joinpoints around which to install timing code around. Complementing this

are new variables that hold the timing measurements; variable names are computed at aspect-weaving time.

The CPU monitoring code installs on top of the timing aspect. This code installs a specifically around the timing points that designate the call to the middle-method code. Instead of using start-to-end elapsed time which would only provide a measure of how long it took to execute a joinpoint, the code measures end-to-end time so that it captures the total time for the application to complete one “round-trip” back to that point. This is compared to the system-reported CPU time to calculate the percentage of CPU used by this process.

The control channel sends data between the two ends of the Infopipe. The data traverses a socket independent of the normal Infopipe data socket both to avoid the overhead of demultiplexing control information and application data and instead piggybacking this functionality on top of the OS demultiplexing which is performed, anyway. Also, separating these two flows of information should improve the general robustness of the application as there is no possibility of errant application data being interpreted as control data or of misleading data being injected as control data somehow.

Finally, there is the SLA aspect. During weaving, it reads an external SLA document which specifies the metrics and tolerances of the values the SLA needs to observe and report. At run time, the SLA reads the CPU usage values and sends them through the control channel to the video; once received, the SLA acts based on the returned value. In the example, the SLA can set a variable to control if and for how long the sender enters `usleep()` to adjust its rate control.

The compiled sample application was run with a “strong” sender, a dual 866MHz Pentium III machine and a “weak,” resource-constrained receiver, a Pentium II 400MHz. Running without any controls on resource usage, the video sender is able to capture on average about 36% of the receiver’s CPU, and CPU usage is furthermore very erratic, as shown in Figure 32.

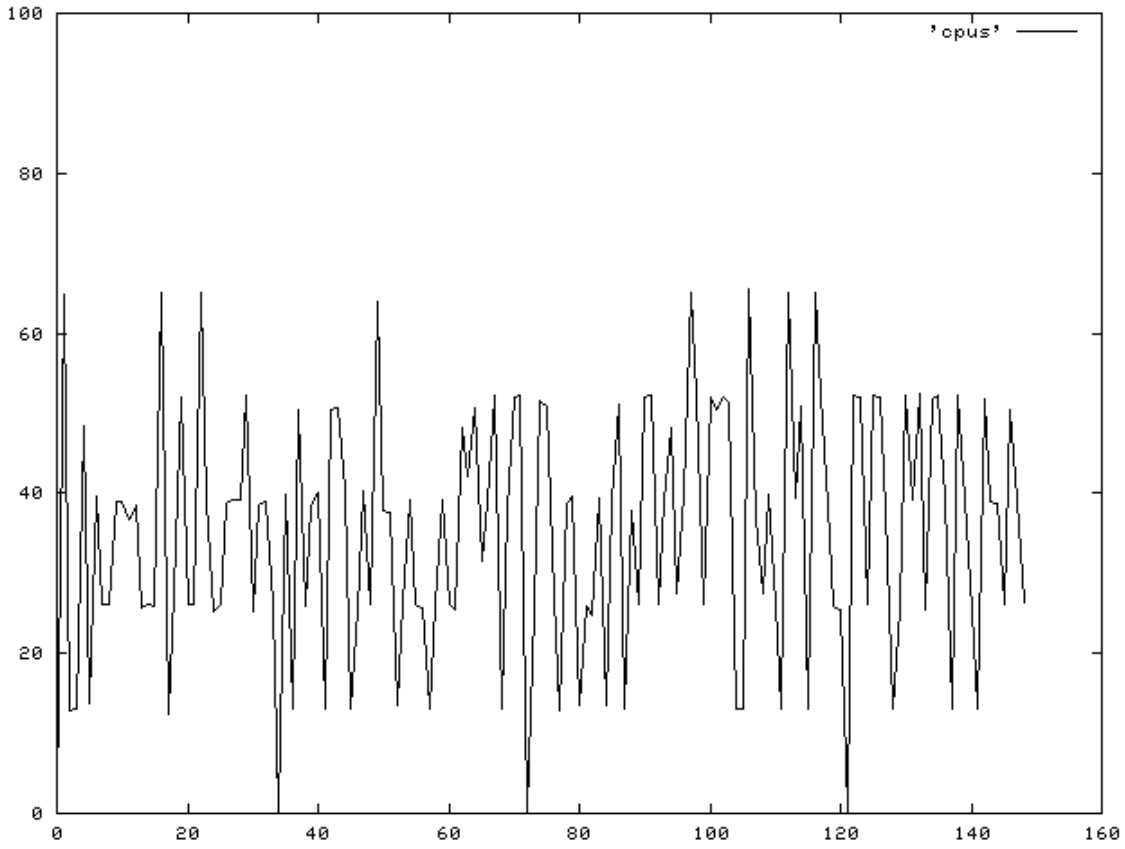


Figure 32. Image receiver CPU usage, no QoS.

Using the CPU control aspects woven in to the application, CPU usage pulls back to generally the target 20±5% range, and the CPU usage is much more predictable as is easily seen in Figure 33. Again, this is only with simple controls. A more sophisticated control and feedback mechanism might provide substantial improvement and would

certainly be required in real-world environment where variable connection fidelity is likely.

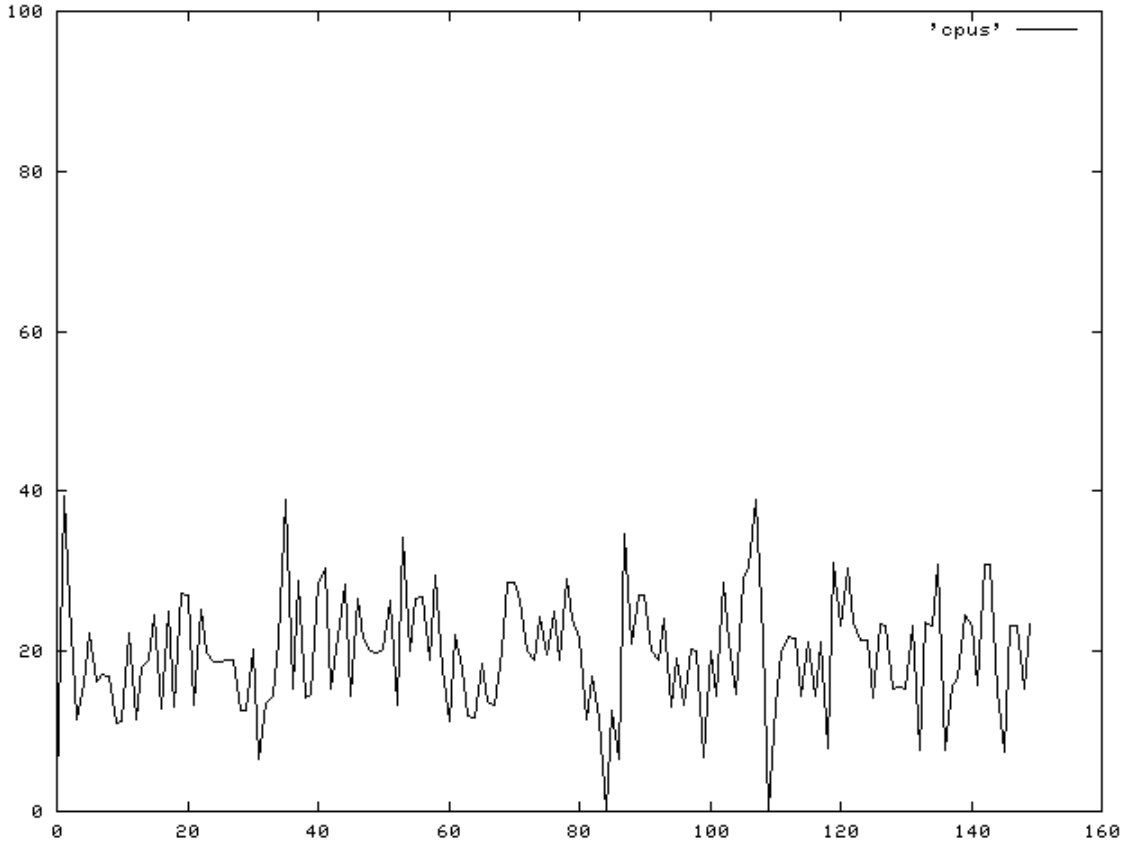


Figure 33. Image receiver CPU usage, with QoS.

3.10.3. Results

After applying each aspect, the ISG dumped the result XIP+ document; it was then stripped the XML, comment lines, and whitespace-only lines. This yielded a monolithic document that contained the source for the entire distributed system equivalent to a concatenation of the generated files minus whitespace and comments. This document allowed counting the number of non-comment lines of code (NCLOC) added by each aspect. For a sample of woven code, see Appendix A which contains an excerpt from the receiver pipe's middle.

Table 14 presents the lines of code added by each aspect. The column "Where" denotes whether the aspect applied to code generated for the sender or the receiver, and the "Lines Added" metric is the number of non-comment lines added. Generally, the aspects add C code, but in the case of the aspects pertaining to the control channel, `control_receiver` and `control_sender` new rules for make were woven in as well.

Table 14. NCLOC Added

Aspect	Where	Lines Added
`control_sender`	sender	117
`sla_sender`	sender	73
`timing`	receiver	50
`control_receiver`	receiver	125
`cpumon`	receiver	14
`sla_receiver`	receiver	55
Total from aspects		434
Base Implementation		976
Base + Aspects		**1410**

So far, aspect files are generally larger than they amount of code they actually generate. However, this tradeoff is appropriate considering the increase in locality of code and reusability (some of these aspects, such as timing and CPU monitoring, have been reused already in another demo). In fact, in examining the files altered or created by the aspects in this application, it is clear that one single-XSLT-file aspect such as the sender-side SLA code can alter four of the generated files and then add two more files of its own. In all, the QoS-aware application is 434 lines longer than the base application that is not QoS aware. Without support from a weaver to help manage code, it would obviously be more difficult to keep track of these 434 lines if they are handwritten into the base set of 18 files versus the six AXpect files.

Table 15 and Table 16 detail exactly which files were affected by the aspect being woven. For the files such as control and SLA, the X in parentheses indicates these files were created by the respective aspect. The number of files altered by each aspect varied greatly. Note that each file may be affected by a variable number of aspects, too. Furthermore, adding a given functionality may not necessarily affect each side of the application (sender vs. receiver) symmetrically.

Table 15. Sender-side files affected.

Aspect / Affected File	Makefile	sender.h	sender.c	PpmOut.h	PpmOut.c	control.h	control.c	sla.c	sla.h
control_sender	X		X		X	(X)	(X)		
sla_sender	X		X			X	X	(X)	(X)

Table 16 Receiver-side files affected.

Aspect / Affected File	Makefile	receiver.h	receiver.c	ppmIn.h	ppmIn.c	control.h	control.c	sla.c	sla.h
timing			X		X				
control_receiver	X		X		X	(X)	(X)		
cpumon			X						
sla_receiver	X		X			X	X	(X)	(X)

3.11. Work Related to Infopipes

Remote Procedure Call (RPC) [10] is the basic abstraction for client/server software. By raising the level of abstraction, RPC facilitated the programming of

distributed client/server applications. Despite its usefulness, RPC provides limited support for information flow applications such as data streaming, digital libraries, and electronic commerce. To remedy these problems, extensions of RPC such as Remote Pipes [42] and AVStreams for CORBA were proposed to support bulk data transfers and sending of incremental results. Still, RPC is usually a poor fit for distributed systems with information flows.

Infopipes have commonality with the dataflow model. Projects which touch on this computing model and code generation include Ptolemy, as mentioned, the SACRES project, and Spidle [2][9][29].

As for QoS and continual queries, there are several continual query projects that are addressing QoS for their systems. However, none have yet demonstrated Quality of Service as it applies to the system beyond the query engine. Aurora [18] supports quality of service within the database engine itself, but it has not addressed quality of service as it applies to the broad system.

STREAM, used for the Linear Road application, is a lightweight query engine compared to Aurora since it does not add the Quality of Service adaptation nor does it require a heavy-duty ACID database to support it and provide persistent storage [4][75].

The Berkeley TelegraphCQ project, like the Aurora project, has also addressed Quality of Service and even distribution or the query server itself [20][60]. Still, Infopipes extend the notion of Quality of Service outside the CQ engine to the entire distributed application.

From a DSL standpoint, Spi/XIP may be compared to Spidle and Streamit. However, Spidle is oriented towards synchronous, single-process applications [29], and

StreamIt aims for streaming DSP applications and processors with grid based architectures [110], [109].

Other AOP projects have taken advantage of DSL patterns various ways. For instance, Bossa applies AOP concepts to scheduling in the kernel [7]. It defines a limited number of pointcuts in the kernel code and then uses an event based AOP model to implement the aspects. However, Bossa does not actually add AOP to a DSL. Instead, Bossa takes the view that the DSL actually implements an aspect describing a single aspect, scheduling, of the Linux kernel. Also, the developers of the ACE+TAO orb used aspect oriented and DSL techniques to expose the real-time functionalities of their orb with contract objects and associated Contract Description Language (CDL) [69]. CDL is limited to monitor and control functions only. However, they then used a second DSL, the Aspect Structure Language (ASL), for applying aspects that mediate interactions between distributed objects [8]. ASL, however, recognizes only a few types of pointcuts that are specific to CORBA development, and application developers can not extend the joinpoint space.

In AOP, XML has been used to capture and manage aspects in the requirements phase [89], and it has been used as the aspect language syntax as in SourceWeave.NET to weave aspect code in the bytecode of the .NET the Common Language Runtime (CLR) [53]. Schonger *et al* proposed XML as a generic markup for describing the abstract syntax trees of general purpose languages and the concept of creating AOP operators for weaving [96]. The results of Infopipes and weaving supports their observation that XML aids AOP experimentation; in addition, Clearwater and ISG implementations explore the use of XML and XSLT use with domain specific languages and code generation.

The AOP and code generation community is actively exploring the new possibilities in combining the two including SourceWeave.NET [53], Meta-AspectJ [117], two-level weaving [46], and XAspects [98].

Before that, The AOP community has worked diligently on weavers for general purpose languages such as Java and C++. This has resulted in tools such as AspectJ, AspectWerkz, JBossAOP, and AspectC [56],[12],[54],[25]. Generally, development of weavers for these platforms requires continual and concerted effort over a fairly long period of time. Other work has tackled separation of concerns for Java through language extensions, such as the explicit programming approach of ELIDE project [15].

DSLs have also often been implemented on top of weavers. Notable in this area is the QuO project, which uses a DSL from which it generates CORBA objects which are called during runtime to be executed at the join point to implement quality of service. However, the QuO project does not weave source code. Instead, it alters the execution path of the application therefore imposes invocation overhead [69]. Bossa uses AOP ideas to abstract scheduling points in OS kernels, but again does not do source weaving; each joinpoint triggers an event and advice registers at events in order to run [7]. Because of the use of aspects in conjunction with DSLs, the XAspects project is studying the use of aspects to implement families of DSLs. Still, this project uses AspectJ as the source weaver and therefore focuses only on Java as the target language [98]. The Meta-AspectJ package also targets enhancing the power of code generators and using code generation plus AOP to reduce complexities for implementing security and persistence [117].

ELIDE is an AOP extension of Java that allows developers to add explicit, named pointcuts at any point in their programs [15]. It differs from the approach in two respects:

first, ELIDE is a Java-specific language extension whereas AXpect works on C and uses generic, pre-existing XML and XSLT tools. Second, ELIDE is general purpose mechanism whereas AXpect is targeted to the domain-specific stub generator and distributed, streaming applications.

Of course, AspectJ and AspectC also implement aspect weavers [56][25], but their weavers and pointcut declarations are closely tied to the implementation structure of the application; therefore, changes to the original source code may break the aspects. AXpect relies instead on explicitly denoted functionality and should be somewhat more robust in that regard.

Finally, Gray et al in [47] propose that AOP techniques be used in specific domains at the level of the domain abstraction as well as at the implementation level. They propose the Embedded Constraint Language (ECL) for creating new domain-specific weavers that process domain models and not implementation source code. This is in contrast to AXpect, which is an implementation level weaver.

3.12. Summary of ISG Research

This chapter outlined the motivation for a high level abstraction called Infopipes for distributed information flow applications such as multimedia streaming and digital libraries, and motivated the use of Clearwater code generation as a technique to implement Infopipes. Infopipes are defined by, and programmed with, a family of domain specific languages called Infopipe Specification Languages (ISL) which is compiled into executable code. The main technical contributions are results from an experimental evaluation of ISG-generated code that demonstrates the advantages of a information flow abstraction such as Infopipe. Measurements of microbenchmarks as

well as application-style benchmarks show that ISG generates high quality code, with minimal additional performance overhead compared to programs hand-written in a general purpose language (C). When compared to SunRPC, results indicate that ISG-generated Infopipe code has comparable latency and significantly better throughput.

The ISG's AXpect AOP engine opens up the generated code for aspect weaving. Writing for AXpect entails aspect specification using a specification, such as WSLA, to parameterize an aspect, and then using XSLT and XPath to write pointcuts and advice that operate on joinpoints explicitly denoted in the code generation templates. Furthermore, code weaving occurs in C source code, a language which does not yet enjoy robust AOP support for the general application space.

Explicit joinpoints are feasible because ISG templates are written in XSLT and because the ISG operates in the specific domain of information flow applications. Since the aspects are also coded using XSLT, joinpoints can efficiently layer to allow aspects to build on functionality and further modularize aspect development. Finally, XSLT allows data retrieval from the XML-based WSLA specification document at "no extra cost" through its support for multiple XML source documents.

The two sample applications demonstrate the ease with which QoS from WSLAs support can be added to an existing domain specific framework using AOP and explicit programming techniques to an existing. These applications show that even adding a relatively simple QoS requirement can entail widespread changes to an application and that those changes can be spread throughout the entire application.

CHAPTER 4
ACCT AND MULINI: CLEARWATER FOR DISTRIBUTED ENTERPRISE APPLICATION MANAGEMENT

4.1. Trends in Enterprise Application Management

New paradigms, such as autonomic computing, grid computing and adaptive enterprises, reflect recent developments in industry and research [52][101][74][43]. This shift is accompanied by increasing complexity in applications, which in turn demands increasing effort to manage these applications. Furthermore, as business intelligence and businesses processes become more enshrined in software, creating "correct" enterprise applications becomes simultaneously more difficult and more important.

These large-scale trends led to the creation of two Clearwater-based code generators. The first, ACCT, attacks the deployment problem for distributed enterprise applications. The second, Mulini, is part of the Elba project works toward automating configuration staging, the pre-production process of validating and verifying application configuration for non-functional properties. In the case of both generators, there are specific challenges that creating tools for the enterprise application domain engenders.

For ACCT, code generation was a natural choice as the problem involved the translation from one specification regime to another, albeit in a non-trivial fashion. For Mulini, the primary reason for adopting it in the Elba project was that staging required the encapsulation of test parameters and variations in some format, and a specification document seemed a natural fit for this purpose. Furthermore, the particular combination and types of challenges faced in enterprise application management, in general, and enterprise application staging, in particular, seemed well suited to a code generation solution.

In fact, there are three specific challenges facing the administrator of enterprise applications. First, the heterogeneity challenge: enterprise applications are often, if not usually, deployed as applications of heterogeneous software deployed onto heterogeneous hardware. Second, the implementation challenge: the tools developed must understand the multiple policy documents and specification encapsulations that accompany today's enterprise applications. Third, the customization challenge: especially in staging, an application must be instrumented and deployed in a manner that is customized from existing production code. That is, an administrator can not simply run a production application in the staging environment and expect to generate meaningful results.

4.2. Some Challenges for Distributed Enterprise Application Management

4.2.1. Heterogeneous Platforms

Heterogeneity for n-tier enterprise applications arises for both hardware and software.

Hardware heterogeneity is, of course, the mixture of various platforms present in the enterprise computing environment on which new applications must be deployed. Contributing factors to hardware heterogeneity include investment preservation, the introduction of new capabilities, and legacy accommodation. Investment preservation occurs when the enterprise re-purposes used, but now underperforming, hardware to new tasks that are less demanding. These performance changes may have been in response to new application features that demanded more functionality or to new end-user demand which the old system could not support. The flip side of capital preservation is the introduction of new hardware. Similarly, it has related reasons – the new hardware may be needed to support new functionality or increased loads of existing applications.

Sometimes the motivation for preserving old systems is simply that legacy applications, tailored to specific “legacy” hardware, such as mainframes, may be wrapped and thereby updated to function as a tier to support n-tier applications. This is especially true if the application relies on an existing database or pool of information that is created over the lifetime of the business.

Enterprises have for some time now operated computing resources in data centers. This naturally supports applications that operate as cluster applications. Such cluster-enabled applications, which include n-tier applications, means that enterprises may leverage the existing mix of hardware and the existing investments to support the new application.

In fact, for an n-tier application, it may be feasible or necessary to support each tier with its own hardware. For instance, the application login tier may be best served by a rack of blades with high-performance processors while the data tier may be best suited to a load-balanced shared cluster of machines which have large RAM capacities and high memory throughput.

The heterogeneous hardware encountered in today’s enterprise is accompanied by similarly heterogeneous software. Just as each tier of an application may require different or specialized hardware for support, it may also require different software. Such differences may be superficial, such as different versions of the same product, or they may be significant enough to require entirely different operating systems installations. A web-based application may very well employ front end pages on a windows machine that interact with backend databases running on high-end Unix systems.

In particular, in staging the problem of heterogeneous software is likely to be encountered as many of the tools used to gauge performance are not likely to be homogeneous with respect to the application being staged. In fact, the heterogeneous tools involved in implementing n-tier applications is likely to demand as many different kinds of performance tools as there are kinds of software in the system. It would be reasonable, for instance, to expect the instrumentation of a system being tested to require a suite of tools to measure software performance in the application logic tier and a separate set of performance tools to measure system response in a database tier. Web service deployments may demand their own set of specialized monitoring agents, again distinct from application and database tools that can evaluate performance with respect to an SLA.

4.2.2. Multiple Input Specifications

The move toward web services as a common deployment paradigm for software continues rapidly. This trend contains within it another problem faced in enterprise systems: there may be multiple specifications that must be incorporated into the design, implementation, and execution of a single application. One can easily imagine a simple web service described by a WSDL, given performance constraints encapsulated in a WSML, and deployed according to a BPEL script into a distributed environment.

Each of these documents also contains information crucial to performing comprehensive and complete staging which must be combined with information from the staging specification (TBL). WSDL information, of course, can be used to ensure the interface correctness of test drivers. WSML, as the codification document for service level agreements, contains within it implicit performance goals the application and

system must meet. Finally, BPEL documents containing deployment information must also be used to calculate deployment dependencies of tools and instrumentation in use during a test. TBL must encode within the differentiators from the production specifications.

Even though web services provide a good generic example of the specification space involved in staging an application, current, non-web-service applications may already have significant specifications associated with them in the guise of location specific policies and constraints, application specific configuration files, and ad hoc deployment scripts. Again, these same documents must be accounted for in the staging of the application to ensure successful staging with a high degree of confidence.

4.2.3. Customizability Requirements

In addition to the difficulties of heterogeneous hardware and multiple specifications, there are also concerns that are not directly addressable. Namely, it is impossible to foresee or implement all possible variations. In short, whatever approach is chosen to support staging must support a degree of customization.

One example of customization is the need to support direct performance instrumentation on application code. Such a case may arise if a particular application instance required finer granularity than system or platform-specific tools may provide.

Naïve approaches to the instrumentation problem might be to require application developers to insert the relevant code when they develop the application beans. However, this suffers from two problems. The developers should be concerned, first, with the functionality of the application beans, not the functionality of testing the application beans. Second, the application developers are then constrained to implement and conform

to a staging interface in addition to their application's functionality interfaces (between beans and from application beans to any backend logic).

Of course, such instrumentation may be pushed into the hands of staging and testing administrators. However, performing such instrumentation by hand will still be error prone and time-consuming and need repetition for each cycle of application staging. Given the needed repetition of the task over multiple program units and multiple staging cycles and further positing that given performance instrumentation *can* be captured in programmatic modules, there is strong case for require customization support from the staging automation software.

In addition, if the tools to support staging are going to support evolution of the staging process and tools, a customization mechanism might provide a low-cost path to exploring solutions to future problems and providing maintenance for the staging system, as well. Such a system may not be "future proof," but at least it might provide "future support."

4.3. ACCT: Clearwater for Application Deployment

With ACCT, the goal is to create "Built-to-Order" systems that operate in these new computing environments. This requires easy, automated application design, deployment, and management tools to address their inherent complexity. The automation system must support creating detailed designs from on user requirements that take into account both operator and technical capability constraints. This automated design is, in itself, a hard problem. Quartermaster Cauldron, addresses the challenge by modeling system components with an object-oriented class hierarchy, the CIM (Common Information Model) metamodel, and embedding constraints on composition within the

models as policies [91]. Then, Cauldron uses a constraint satisfaction approach to create system designs and deployment workflows. However, these workflows and designs are expressed in system-neutral Managed Object Format (MOF).

4.3.1. Automated Configuration Design Environment

MOF workflows typically involve multiple systems and formats that have to be dealt with in order to deploy a complex system. For example, deploying a three-tier e-commerce solution in a virtualized environment may involve interactions with blade servers, VMWare/Virtual Servers, multiple operating systems, service containers for web servers, application servers, databases, before, finally, executing clients scripts. This problem of translating generic design in a system independent format (*e.g.*, MOF) to the multiple languages/interfaces demanded by the system environment is thus nontrivial.

The main contribution of the ACCT research is a generic mechanism for translating design specifications written in a system independent format into multiple and varied deployment environments. Translation between the two models is non-trivial and significant result for two reasons.

First, Cauldron and SmartFrog operate on differing system models which are quite dissimilar in some aspects; the translation is not a straightforward one-to-one mapping. Cauldron emits MOF, but SmartFrog requires a SmartFrog workflow document plus a set of Java class definitions. Also, Cauldron generates a deployment plan consisting of pairwise dependencies between application components whereas SmartFrog needs a complete workflow specification of all dependencies.

Second, the ACCT design is deliberately generic to accommodate the multiple input and output formats encountered across multiple design and deployment

environments. ACCT accepts MOF-based design specifications of CIM instance models and converts them into input specifications for SmartFrog, a high-level deployment tool [99]. SmartFrog uses on a high-level specification language and Java code to install, execute, monitor, and terminate applications.

At the highest level of abstraction, automated design tools offer streamlined and verified application creation. Quartermaster is an integrated tool suite built around MOF to support automated design of distributed applications at this high level of abstraction [94]. Cauldron, one of its key components, supports applying policies and rules at design-time to govern composition of resources [92]. Cauldron's constraint satisfaction engine can generate system descriptions that satisfy these administrative and technical constraints in addition to the deployment constraints handled by ACCT. Since each component of an application often depends on prior deployment of other components or completion of other components' work, deployment is non-trivial.

A deployment model is built as a MOF Activity comprised of a number of sub-activities. Each of these activities has a set of constraints to meet before execution and also parameters that must receive values. At design time, Cauldron generates configuration templates and also pairwise deployment dependencies between deployment activities. Between any pair of activities, there are four possible synchronization dependencies.

SS (Start-Start) – activities must start together; a symmetric, transitive dependency.

FF (Finish-Finish) –activities must finish together (synchronized); also a symmetric, transitive dependency.

FS (Finish-Start) – predecessor activity must complete before the successor activity is started, *i.e.*, sequential execution. This dependency implies a strict ordering, and the MOF must assign either the antecedent or the dependant role to each activity component.

SF (Start-Finish) – predecessor activity is started before the successor activity is finished. Similar observations on its properties follow as from FS. (As an SF example, consider producer-consumer relationships in which the producer must create a communication endpoint before the consumer attempts attachment.)

Cauldron, however, is solely a design tool and provides no deployment tools, which require software that initiate, monitor, and kill components in a distributed environment.

4.3.2. Automated Application Deployment Environment

Automated deployment tools serve to ameliorate the laborious process of preparing, starting, monitoring, stopping, and even post-execution clean-up of distributed, complex applications. SmartFrog is an open-source, LGPL framework that supports such service deployment and lifecycle management for distributed Java applications [45][99]; it has been used on the Utility Computing model for deploying rendering code on demand and has been ported to PlanetLab [81]. Expertise gained applying SmartFrog to grid deployment [19] is being used in the CDDLM standardization effort currently underway.

Conceptually, SmartFrog comprises 1) a component model supporting application-lifecycle operations and workflow facilities, 2) a specification language and validator for these specifications, and 3) tools for distribution, lifecycle monitoring, and control. The main functionalities of SmartFrog are as follows:

Lifecycle operations – SmartFrog wraps deployable components and transitions them through their life phases: *initiate, deploy, start, terminate,* and *fail*.

Workflow facilities – Allows flexible control over configuration dependencies between components to create workflows. Examples: *Parallel, Sequence,* and *Repeat*.

SmartFrog runtime – Instantiates and monitors components; provides security. The runtime manages interactions between daemons running on remote hosts. It provides an event framework to send and receive events without disclosing component locations.

SmartFrog's specification language features data encapsulation, inheritance, and composition which allows system configurations to be incrementally declared and customized. In practice, SmartFrog needs three types of files to deploy an application:

- Java `interface` definitions for components. These serve analogously to the interface exposure role of the C++ header file and `class` construct.
- Java source files that implement components as objects. These files correspond one-to-one with the above SmartFrog component descriptions.
- A single instantiation and deployment file, in a SmartFrog specific language, defining the parameters and proper global deployment order for components.

There are several obstacles to translating Cauldron to SmartFrog. First, there is the syntax problem; Cauldron generates MOF, but SmartFrog requires a document in its own language syntax as well as two more types supporting of Java source code.

Obviously, this single MOF specification must be mapped to three kinds of output files, but neither SmartFrog nor Quartermaster supports deriving Java source from the design documents. Finally, Cauldron only produces *pairwise* dependencies between deployment activities; SmartFrog, on the other hand, needs dependencies over the entire set of deployment activities to generate a deployment workflow for the system. It is the job of the ACCT generator to extract the deployment data from the Cauldron output, resolve the conflicts above, and to create SmartFrog and Java code for deployment execution.

4.3.3. Translating from Design Specifications to Deployment Specifications

ACCT, being a Clearwater generator, supports extensible specifications, pliable code generation, and output customization; its output is a fully parameterized, executable deployment specification. Despite the fact that both are Clearwater generators, ACCT and ISG do have some differences in their implementations. First described is ACCT's design and the implementation and then the mapping approach needed to resolve mismatches between the design tool output (MOF) and deployment tool input (SmartFrog).

The code generation consists of three phases which are illustrated in Figure 34. In the first phase, MOF-to-XML, ACCT reads MOF files and compiles them into a single XML specification, XMOF, using a modification of the publicly available WBEM Services' CIM-to-XML converter [112]. While this diagram shows only Java and SmartFrog support, recent development efforts added script-based deployment and research is proceeding into other deployment formats as well.

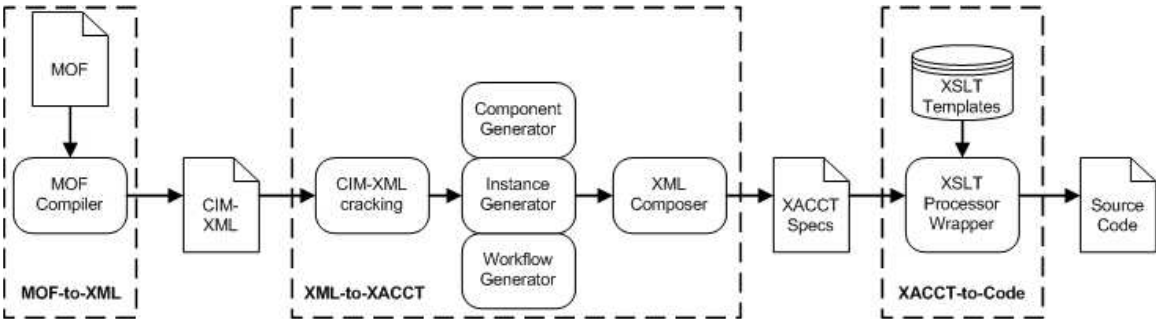


Figure 34. ACCT.

In phase two, XML-to-XACCT, XMOF is translated into a set of XACCT documents, the intermediate XML format of the ACCT tool. During this transformation, ACCT processes XMOF in-memory as a DOM tree and extracts three types of Cauldron-embedded information: components, instances, and deployment workflow. Each data set is processed by a dedicated code generator written in Java. The component generator creates an XML component description, the instance generator produces an XML fragment containing component attributes and values for the deployed components, and the workflow generator computes a complete, globally-ordered workflow expressed in XML. These generated structures are passed to an XML composer which performs rudimentary type checking (to ensure instances are only present if there is also a class), and re-aggregates the XML fragments back into a whole XML documents. This may result in multiple XACCT component description documents, but there is only one instantiation+workflow document which contains the needed data for a single deployment.

Finally, in the third phase ACCT forwards each XACCT component description, instantiation, and workflow document to the XSLT processor. The XSLT templates

detect the XACCT document type and generate the appropriate files (SmartFrog or Java) which are written to disk.

XACCT allows components, configurations, constraints, and workflows from input languages of any resource management tool to be described in an intermediate representation. Once an input language is mapped to XACCT, the user merely creates an XSLT template for the XML-to-XACCT phase to perform the final mapping of the XACCT to a specific target language. Conversely, one need only add a new template to support new target languages from an existing XACCT document.

Purely syntactic differences between MOF and SmartFrog's language can be resolved using solely XSLT, and the first version of ACCT was developed in XSLT alone. However, because the XSLT specification version used for ACCT had certain limitations, ACCT incorporates Java pre- and post- processing stages. This allows ACCT to compute the necessary global SmartFrog workflows from the MOF partial workflows and to create multiple output files from a single ACCT execution.

The overall system ordering of the workflows derives from the Cauldron computed partial synchronizations encoded in the input MOF. As mentioned, MOF defines four types of partial synchronization dependencies: SS, FF, SF, and FS. To describe the sequential and parallel ordering of components which SmartFrog requires, these partial dependencies are mapped via an event queue model with an algorithm that synchronizes activities correctly [91][114][115].

4.3.4. Demo Application and Evaluation

This section provides a working example of how Cauldron-ACCT-SmartFrog toolkit operates from generating the system configurations and workflow, translating both

into the input of SmartFrog, and then automatically deploying distributed applications of varying complexity. Next is the description of the 1-, 2-, and 3-tier example applications and system setup employed for the experiments. Following the example presentation, toolkit evaluation is performed by comparing the deployment execution time of SmartFrog and automatically generated deployment code to manually written deployment scripts.

4.3.4.1. Experiment Scenario and Setup

ACCT was evaluated by employing it on 1-, 2-, and 3-tier applications. The simple 1- and 2-tier applications provide baselines for comparing a generated SmartFrog description to hand-crafted scripts. The 3-tier testbed comprises web, application, and database servers; it is a small enough size to be easily testable, but also has enough components to illustrate the power of the toolkit for managing complexity. Table 17, below, lists each scenario's components.

Table 17. Components of 1-, 2-, and 3-tier applications

Scenario	Application	Components
1-tier	Static web page	Web Server : Apache 2.0.49
2-tier	Web Page Hit Counter	Web Server : Apache 2.0.49 App. Server : Tomcat 5.0.19 Build System: Apache Ant 1.6.1
3-tier	iBATIS JPetStore 4.0.0	Web Server : Apache 2.0.49 App. Server : Tomcat 5.0.19 DB Server : MySQL 4.0.18 DB Driver : MySQL Connector to Java 3.0.11 Build System : Apache Ant 1.6.1 Others : DAO, SQLMap, Struts

SmartFrog 3.04.008_beta was installed on four 800 MHz dual-processor Dell Pentium III machines running RedHat 9.0; one SmartFrog daemon runs on each host.

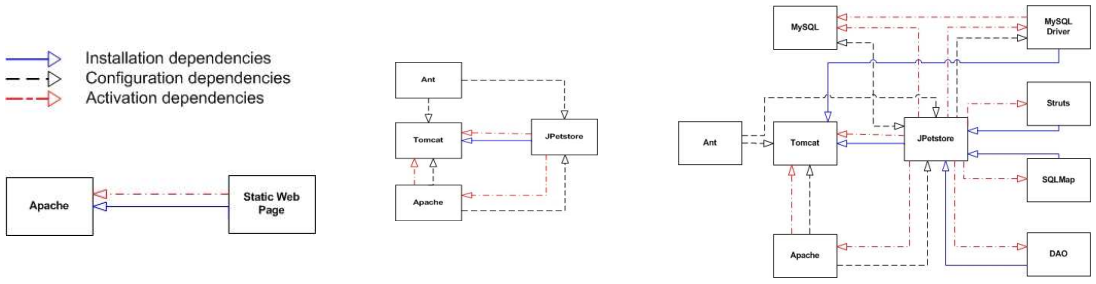
In the 1-tier application, Apache was deployed as a standalone web server, and confirmed successful deployment by visiting a static web page. The evaluation used two machines: the first for the web server and a second to execute the generated SmartFrog workflow.

In the 2-tier Hit Counter application used Apache and the Tomcat application server with the Ant build system. Each tier specified for deployment to a separate host. To verify the 2-tier deployment, an operator visited the web page multiple times to ensure it recorded page hits. The application simply consists of a class and a `jsp` page. The 2-tier evaluation required three machines. As in the 1-tier test, one machine was used to run the deployment script; then, another single machine was dedicated to each deployed tier (Apache; Ant and Tomcat).

Finally, the 3-tier application was the iBATIS JPetStore, a ubiquitous introduction to 3-tier programming. The 3-tier application evaluation used four machines. Again, one machine was dedicated for each tier (Apache; Tomcat, JPetStore, Ant, MySQL Driver, Struts; MySQL DB) and fourth machine ran the SmartFrog workflow.

Figure 35 illustrates the dependencies of components in each testbed. There were three types of dependencies considered in the experiment; installation dependency, configuration dependency, and activation dependency. The total number of dependencies in each testbed is used as the level of the complexity. In the figure, 1-, 2-, and 3-tier testbeds are considered as simple, medium, and complex cases respectively. Intuitively, the installation, configuration, and activation dependencies of each component in each

testbed must be sequenced. For instance, the Apache configuration must start after Apache installation completes, and Apache activation must start after Apache configuration completes. (For space, these dependencies are omitted from the figure.)



a) b) c)

Figure 35. Dependency diagrams of (a) 1-tier, (b) 2-tier, and (c) 3-tier application.

The 1-, 2-, and 3-tier applications were modeled in Quartermaster with the Cauldron module created the configurations and deployment workflows. The resultant MOF files were fed into ACCT and yielded a set of Java class files, SmartFrog component descriptions, and a SmartFrog instances+workflow specification for each application tested. Figure 36 on the following page illustrates the transformation process as ACCT translates the MOF file of the 3-tier application to intermediate XACCT and then finally to a SmartFrog description. For demonstration, the FS dependency between the Tomcat installation and MySQLDriver installation is highlighted and how this information is carried through the transformation process.

```
instance of LogicalServer {
   Id = "Tomcat_LS1";
   Caption = "Tomcat Logical Server";
   Description = "Logical Server for Tomcat ";
   IpAddress = "130.207.5.228";
   HostName = "artemis.cc.gatech.edu";
};
instance of LogicalServerInLogicalApplication  {
   LogicalApplication = "Tomcat\";
   LogicalServer =Tomcat_LS1\";
};
instance of LogicalApplication {
   Id = "Tomcat";
   Version = "5.0.19";
   Caption = "Tomcat";
   Description = "Tomcat application Server";
};
instance of LogicalApplication {
   Id = "MySQLDriver";
   Version = "3.0.11";
   Caption = "MySQLDriver";
   Description = "MySQL driver";
};
instance of Activity {
   Id = "Tomcat_Installation";
   ActivityType = "script";
};
instance of Activity {
   Id = "Tomcat_Installation";
   ActivityType = "script";
};
Instance of ActivityPredecessorActivity {
   DependenceType=”Finish-Start”;
   AntecedentActivity=”Tomcat_Installation”;
   DependentActivity=”MySQLDriver_installation”;
};
```

a) MOF

```
<Instance Name="Tomcat" Class="LogicalApplication">
   <Variable Name="Id"Type="string">Tomcat</Variable>
   <Variable Name="Version"Type="string">
      5.0.19</Variable>
   <Variable Name="Entity" Type="string">
      Activity_Tomcat_Installation</Variable>
   <Variable Name="Host" Type="string">
      artemis.cc.gatech.edu</Variable>
</Instance>
   <Workflow>
      <Work Type=”Execution”></Work>
      <Work Type=”EventSend”>
         <To> MySQLDriver_Installation</To></Work>
      <Work Type=”Terminate”>
        Tomcat_Installation </Work>
   </Workflow>
<Instance  Name="MySQLDriver"
                 Class="LogicalApplication">
   <Variable Name="Id" Type="string">
      MySQLDriver</Variable>
   <Variable Name="Version" Type="string">
      3.0.11</Variable>
   <Variable Name="Entity" Type="string">
      Activity_MySQLDriver_Installation</Variable>
   <Variable Name="Host" Type="string">
      demeter.cc.gatech.edu</Variable>
</Instance>
<Workflow>
   <Work Type=”OnEvent”>
      <From> Tomcat_Installation</From> </Work>
   <Work Type=”Execution”></Work>
   <Work Type=”Terminate”>
      MySQLDriver_Installation</Work>
</Workflow>
```

b) XACCT

```
sfProcessComponentName  "Tomcat_Installation";
LogicalApplication_Tomcat  extends  LogicalApplication {
      Id   "Tomcat";
      Version   "5.0.19";
      Activity LAZY ATTRIB
Activity_Tomcat_Installation;
      sfProcessHost   "artemis.cc.gatech.edu";
}
Activity_Tomcat_Installation  extends  Activity {
      Id   "Tomcat_Installation";
      Entity   LAZY ATTRIB LogicalApplication_Tomcat;
}
-- extends EventSend {
      sendTo eventQueue:queue_Tomcat_Ignition;
      event "Activity_Tomcat_Installation_FS";
}
-- extends Terminator {
      kill eventQueue:queue_Tomcat_Installation;
}
sfProcessComponentName "MySQLDriver_Installation";
-- extends OnEvent {
     registerWith queue_MySQLDriver_Installation ;
     Activity_Tomcat_Installation_FS extends DoNothing
}
LogicalApplication_MySQLDriver extends LogicalApplication
{
      Id   "MySQLDriver";
      Version   "3.0.11";

ActivityLAZYATTRIBActivity_MySQLDriver_Installation;
      sfProcessHost   "demeter.cc.gatech.edu";
}
Activity_MySQLDriver_Installation extends Activity {
      Id   "MySQLDriver_Installation";
     Entity  LAZY ATTRIB
LogicalApplication_MySQLDriver;
}
-- extends Terminator {
     kill eventQueue:queue_MySQLDriver_Installation;
}
```

c) SmartFrog

Figure 36. (a) MOF, (b) Intermediate XML, and (c) SmartFrog code snippets. The solid line box indicates the FS workflow between Tomcat and MySQLDriver applications. Others indicate configurations. Clearly, MOF offers superior understandability for a deployment scenario as compared to the SmartFrog specification. As Vanish et al showed in [108], automating deployment via SmartFrog is generally superior in performance and more maintainable when compared to manual or ad hoc scripted solutions.

4.3.4.2. Experimental Result

The metric for evaluating the 1-, 2-, and 3-tier testbeds is deployment execution time as compared to manually written scripts. Each SmartFrog and script was executed 30 times each for each tier application and report the average. Figure 37 shows that for simple cases (1- and 2-tier) SmartFrog took marginally longer when compared to the scripts based approach because SmartFrog daemons running in the Java VM impose extra costs when loading Java classes or engaging in RMI communication. The trend favors SmartFrog as the time penalty of the medium case becomes less (in absolute and relative terms) and for the complex case, SmartFrog took less time than the scripts based approach.

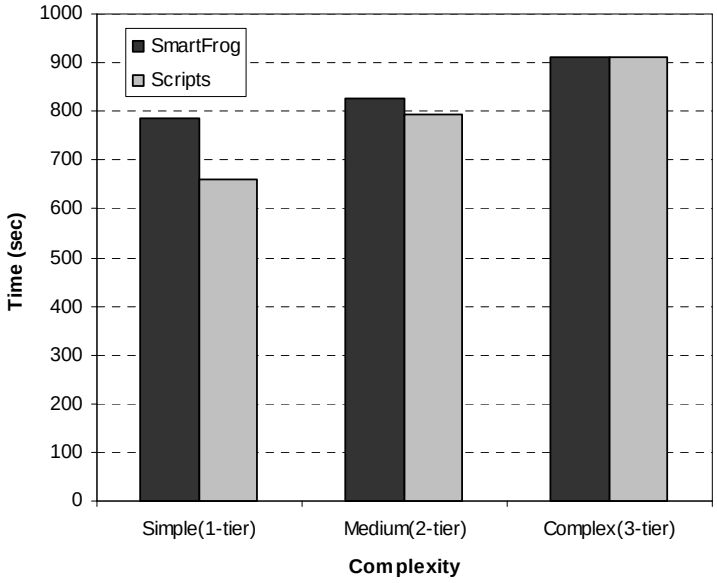


Figure 37. Deployment Time using SmartFrog and scripts as a function of the complexity.

In the complex case, SmartFrog was able to exploit concurrency between application components since it had a computed workflow. The simple and medium cases contain fewer concurrent dependencies than the 3-tier case.

Nevertheless, in all cases the toolkit retains the important advantage of an automatically generated workflow, while in scripts based approach, system administrators must manually control the order of installing, configuration, and deployment.

4.4. Mulini: Clearwater for Application Staging

4.4.1. Motivation, the Elba Project

Managing the growing complexity of large distributed application systems in enterprise data center environments is an increasingly important and increasingly expensive technical challenge. While design, staging, deployment, and in-production activities such as application monitoring, evaluation, and evolution are complex tasks in themselves, staging in particular engenders unique challenges first because of its role linking development and deployment activities and second because of the need for staging to validate both functional and extra-functional properties of an application. This process is complicated by multiple iterations, each much shorter in duration than expected for the application's in production runtime.

Currently, developers and administrators perform staging tasks manually, or they use scripts to achieve limited *ad hoc* automation. This section focuses on the automation of performance testing for extra-functional validation (of an automatically generated configuration) during the staging process; this offers a way to detect and prevent serious problems that may arise before new configurations are deployed to their production

environment. The approach ties together the production tools into a staging toolkit, and their associated production policy documents are translated via a staging specification language and associated code generator.

Staging is a natural choice for data center environments where sufficient resources are available for adequate evaluation. It allows developers and administrators to tune new deployment configuration and production parameters under simulated conditions before the system goes "live". However, traditional staging is usually approached in a manual, complex, and time consuming fashion. In fact, while the value of staging increases with application complexity, the limitations inherent to manual approaches tend to decrease the possibility of effectively staging that same complex application.

Furthermore, increasing adoption of Service-Level Agreements (SLAs) that define requirements and performance also complicates staging for enterprise-critical, complex, and evolving applications; again, the limitations of the manual approach become a serious obstacle. SLAs provide quantitative metrics to gauge adherence to business agreements. For service providers, the staging process allows them to "debug" any performance (or other SLA) problems *before* production and thereby mitigate the risk of non-performance penalties or lost business.

This section describes the Elba project, the goal of which is to provide a thorough, low-cost, and automated approach to staging that overcomes the limitations of manual approaches and recaptures the potential value of staging. The main contribution is the Clearwater-based Mulini staging code generator which uses formal, machine-readable information from SLAs, production deployment specifications, and a test-plan

specification to automate the staging phase of application development. Mulini-generated code ties existing deployment tools together with new staging-specific information and instrumentation.

The chapter summarizes implementation of Mulini and includes an early evaluation of Mulini generated staging code for staging a well-known application, TPC-W [72], which includes performance-oriented service level requirements. By varying the specifications, Mulini can generate and compare several configurations and deployments of TPC-W of varying costs. Note that TPC-W is used as an illustrative complex distributed application for the Elba/Mulini automated staging tools and process, not necessarily as a performance measure of the hardware/software stack. For this reason, TPC-W is referred to as an "application" rather than its usual role as a "benchmark."

4.4.2. Requirements in Staging

Staging is the pre-production testing of application configuration with three major goals. First, it verifies functionality, *i.e.*, the system does what it should. Second, it verifies the satisfaction of performance and other quality of service specifications, *e.g.*, whether the allocated hardware resources are adequate. Third, it should also uncover *over*-provisioned configurations. Large enterprise applications and services are often priced on a resource usage basis. This question involves some trade-off between scalability, unused resources, and cost of evolution (discussed briefly in Section 4.4.8). Other benefits of staging, beyond the scope of this paper, include the unveiling of other application properties such as its failure modes, rates of failure, degree of administrative attention required, and support for application development and testing in realistic configurations.

These goals lead to some key requirements in the successful staging of an application. First, to verify the correct functionality of deployed software on hardware configuration, the staging environment must reflect the reality of the production environment. Second, to verify performance achievements the workload used in staging must match the service level agreement (SLA) specifications. Third, to uncover potentially wasteful over-provisioning, staging must show the correlation between workload increases and resource utilization level, so an appropriate configuration may be chosen for production use.

These requirements explain the high costs of a manual approach to staging. It is non-trivial to translate application and workload specifications accurately into actual configurations (requirements 1 and 2). Consequently, it is expensive to explore a wide range of configurations and workloads to understand their correlation (requirement 3). Due to cost limitations, manual staging usually simplifies the application and workload and runs a small number of experiments. Unfortunately, these simplifications also reduce the confidence and validity of staging results.

Large enterprise applications tend to be highly customized "built-to-order" systems due to their sophistication and complexity. While the traditional manual approach may suffice for small-scale or slow-changing applications, built-to-order enterprise applications typically evolve constantly and carry high penalties for any failures or errors. Consequently, it is very important to achieve high confidence during staging, so the production deployment can avoid the many potential problems stemming from complex interactions among the components and resources. To bypass the

difficulties of manual staging, it seems reasonable to advocate an automatic approach for creating and running the experiments to fulfill the above requirements.

4.4.3. Staging Steps

In automating the staging process, staging is divided into three steps: design, deployment, and the actual test execution. The automation tools of the first two steps produce automatically generated and deployable application configurations for the staging environment. The third step, execution, is to generate and run an appropriate workload on the deployed configuration and verify the functionality, performance, and appropriateness of the configuration.

In the first step, design, the entire process starts with a machine-readable specification of detailed application design and deployment. Concretely, this has been achieved by Cauldron [92], an application design tool that generates system component specifications and their relationships in the CIM/MOF format (Common Information Model, Managed Object Format). Cauldron uses a constraint satisfaction approach to compute system designs and define a set of workflow dependencies during the application deployment.

The second step in the automated staging process is the translation of the CIM/MOF specification into a concrete configuration. Concretely, this is achieved by ACCT (Automated Composable Code Translator) already described.

4.4.4. Automated Staging Execution

The third step of staging is the automation of staging execution. Automated execution for staging requires three main components: (1) a description mapping the application to the staging environment, (2) the input to the application – the workload

definition, and (3) a set of application functionality and performance goals defined on the workload.

The application description (first component) can be borrowed or computed from the input to the first and second steps in which the application has been formally defined. However, environment dependent parameters may make re-mapping the execution parameters of an application from a deployment to staging environment a non-trivial task. For instance, required location sensitive changes obviously include location strings for application components found in the design documents. On the other hand, non-obvious location dependencies also exist within the application such as references to services the application may require to execute successfully, like an ORB (Object Request Broker) naming service or URIs.

For the staging workload definition (the second component), it is advantageous to reuse the production workload if available. The use of a similar workload increases the confidence in staging results. Also, by mapping the deployment workload into the staging environment automatically, the study of the correlation between workload changes and resource utilization in different configurations is facilitated because the low-cost, repeatable experiments encourage the testing of multiple system parameters in fine-grain steps. The repeatability offered by an automated system provides confidence in the behavior of the application to a presented workload as the application evolves during development and testing.

The third component is specification and translation of application functionality and performance goals into a set of performance policies for the application. This is a “management task” and the main information source is the set of Service Level

Agreements (SLAs). Typically, SLAs explicitly define performance goals such as “95% of transactions of Type 1 will have response time under one second”. These goals, or Service Level Objectives, can serve as sources for deriving the monitoring and instrumentation code used in the staging process to validate the configuration executing the intended workload. Beyond the customer-oriented SLAs, there may also be defined performance requirements that derive not from customer demand but from internal policies of the service provider.

The automated translation processes of each single component and of all three components are significant research challenges. In addition to the typical difficulties of translating between different levels of abstraction, there is also the same question of generality applicable to all application-focused research projects: how to generalize the results and apply the techniques to other applications.

4.4.5. Elba Approach and Requirements

As summarized, staging tools must process three major components when automating staging: the application, the workload, and performance requirements. One of the main research challenges is the integrated processing of these different specifications through the automated staging steps. The Elba approach is to create and define an extensible specification language called TBL (the testbed language) that captures the peculiarities of the components as well as the eventual target staging environment. The incremental development of TBL and associated tools is the cornerstone of Elba.

Research goals for the specification of applications and their execution environments include: automated re-mapping of deployment locations to staging locations; creation of consistent staging results across different trials; extensibility to

many environments and applications. Research goals on the evaluation of application quality of service (QoS) include:

- Developing appropriate QoS specifications and metrics that capture SLAs as well as other specification methods.
- Instrumentation for monitoring desired metrics. This automates the staging result analysis.
- Matching metrics with configuration resource consumption. An SLA defines the customer-observable behavior (*e.g.*, response time of transactions) but not corresponding underlying system resource usage (*e.g.*, CPU usage of specific nodes).
- Maintaining a low run-time overhead (*e.g.*, translation and monitoring) during automated staging.
- Generating reports that summarize staging results, automatically identifying bottlenecks as appropriate.

4.4.6. Summary of Tools

These goals must be met in Elba's context supporting the staging process with cyclical information and control flow, shown in Figure 38. A cyclical view of staging allows feedback from execution to influence design decisions before going to production. The figure shows how new and existing design tools can be incorporated in the staging process if their specification data can be transformed to support staging. Specifically, the design builds on Cauldron, which maps software to hardware, and ACCT+S which maps deployment declarations into the space of deployment engines, and these tools cooperate with the staging specific tool Mulini.

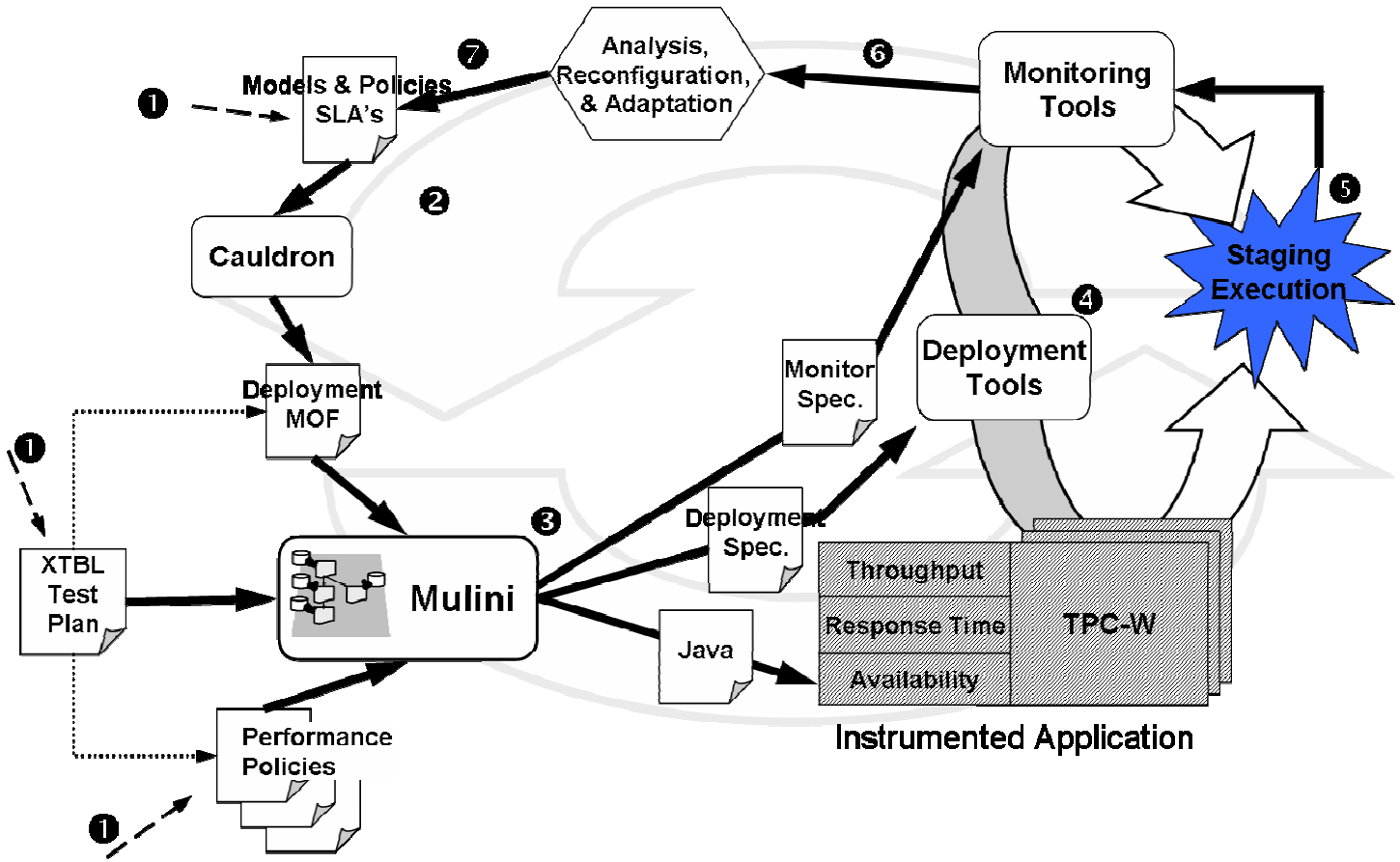


Figure 38. The goal of the Elba is to automate staging by using data from high-level documents. The staging cycle: 1) Developers provide design-level specifications and policy as well as a test plan (XTBL). 2) Cauldron computes a deployment plan for the application. 3) Mulini generates a staging plan from its inputs. 4) Deployment tools deploy the application and monitoring tools to the staging environment. 5) The staging is executed. 6) Data from monitoring tools is gathered for analysis. 7) After analysis, developers adjust deployment specifications or possibly even policies and repeat the process.

First, for provisioning and application description, the Cauldron design tool provides a constraint-based solver that interprets CIM description scenarios to compute application deployment specifications [92]. This allows application developers to leverage the inherent parallelism of the distributed environment while maintaining correctness guarantees for startup. For example, the database data files are deployed before the database is started and the image files are deployed before the application server is started. Future Elba development will extend Cauldron development to incorporate SLA information in the provisioning process. This move will allow Cauldron's formal constraint engines to verify the SLAs themselves and incorporate SLA constraints into the provisioning process. These SLAs can then be converted into XML-based performance policy documents for Mulini, as illustrated in Figure 38.

The new component described in this paper is the Mulini code generator for staging; it implements the second and third steps, workload definition and specification translation, from information contained in TBL. It includes ACCT+S, an automated deployment generator tool that generates application configuration from the CIM/MOF specification generated by Cauldron. ACCT+S output is executed by deployment tools such as SmartFrog.

4.4.7. Code Generation in Mulini

As mentioned, the staging phase for an application requires three separate steps: design, deployment, and execution. Again, requirements for automated design are fulfilled by Quartermaster/Cauldron and deployment is fulfilled by ACCT. Mulini's design wraps the third step, execution, with deployment to provide an automated approach to staging.

Mulini has four distinct processing phases: specification integration, code generation, code weaving, and output. In the current, early version, these stages are at varying levels of feature-completeness. The following describes features to be included in near term releases, and then the end of this section briefly describes current implementation status. Figure 39 illustrates the generator and its components.

In specification integration, Mulini accepts as input an XML document that contains the basic descriptors of the staging parameters. This step allows Mulini to make policy-level adjustments to specifications before their processing by policy driven tools such as ACCT+S. The document, XTBL, contains three types of information: the target staging environment deployment information to which should be re-mapped, a reference to a deployment document containing process dependencies, and references to performance policy documents containing performance goals.

Mulini integrates these three documents into a single XTBL+ document. XTBL+ is organized with the same structure as the XTBL, but leverages XML's extensibility to include the deployment information and performance requirements information. The weaving process for these documents:

1. Load, then perform XML parsing, and construct a DOM tree of the XTBL document. This becomes the core for a new XTBL+ document.
2. Retrieve all references to deployment documents (XMOF documents) from the XTBL document. There may be more than one deployment document since deployment of resource monitors may be specified separately from deployment of application components.

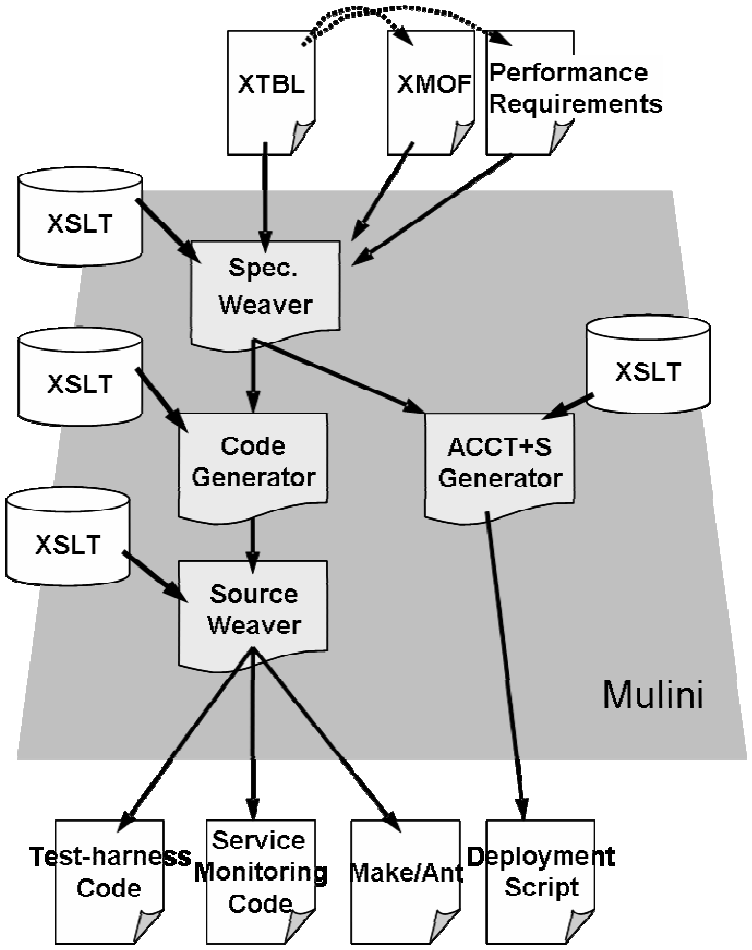


Figure 39. The grey box outlines components of the Mulini code generator. Initial input is an XTBL document. The XTBL is augmented to create an XTBL+ document used by the two generators and the source weaver. The Specification weaver creates the XTBL+ by retrieving references to the performance requirements and the XMOF files and then weaving those files.

3. Load any referenced deployment documents and incorporate their code onto the XTBL+ document. At this point, the deployment directives are re-targeted to the staging environment from the deployment environment, *e.g.*, machine names and URLs must be remapped. In the XTBL document, each deployable unit is described with an XMOF fragment; each of the targeted deployment hardware is also described with an XMOF fragment.
4. Load the performance requirements documents. Mulini maps each performance requirement mapped onto its corresponding XMOF deployment component(s). This yields an integrated XTBL+ specification.

Figure 50 of Appendix B illustrates the three source documents and the woven XTBL+ result.

Following the specification weaving, Mulini generates various types of source code from the XTBL+ specification. To do so, it uses two sets of code generators. The first of these code generators is the ACCT generator which has been used to generate SmartFrog deployments of applications. To enhance flexibility for its use in Mulini, ACCT was extended to support script-based deployments and so refer to it here as ACCT+S. ACCT+S accepts the XTBL+, extracts relevant deployment information, generates the deployment scripts, and writes them into files.

The second code generator creates staging-phase application code, which for TPC-W is Java servlet code, shell scripts for executing monitoring tools, and `Makefile`s. The TPC-W code includes test clients that generate synthetic workloads, application servlets, and any other server-side code which may be instrumented. If source

code is available, it can be added to Mulini's generation capabilities easily, the process for importing a TPC-W servlet into Mulini for instrumentation is described later in this section. Mulini, in this phase, also generates a master script encapsulating the entire staging execution step, compilation, deployment of binaries and data files, and execution commands, which allows the staging to be executed by a single command.

Staging may require the generation of instrumentation for the application and system being tested. Mulini can generate this instrumentation: it can either generate tools that are external to and monitor each process through at the system level (*e.g.*, through the Linux `/proc` file system), or it may generate new source code in the application directly.

The source weaver stage of Mulini accomplishes the direct instrumentation of source code by weaving in new Java code that performs fine grain instrumentation on the base code. The base code may be either generic staging code generated from the specification inputs, or it may be application-specific code that has been XSLT-encapsulated for Mulini weaving. Source weaving is similar to that of the AXpect weaver.

Practically, of course, this means that the application code must be included in the generation stream. Fortunately, the use of XML enables this to be done quite easily. Source code can be directly dropped into XSLT documents and escaping can be automatically added for XML characters with special meaning, such as ampersand and '<'. At aspect weaving time, this code can be directed to be emitted, and semantic tags enable the weaver to augment or parameterize the application code.

As mentioned earlier, the instrumentation code may derive from SLAs that govern the service expectations of the deployed application. These service level documents contain several parts. First, they name the parties participating in the SLA as well as its dates of enforcement. Then, they provide a series of service level objectives (SLOs). Each SLO defines a metric to be monitored, location at which it is to be measured, conditions for monitoring (start and stop), and any subcomponents that comprise that metric. For instance, the "ResponseTime" metric comprises response time measurements for each type of interaction in the TPC-W application.

At this time, most of the Mulini functionality has been implemented. This includes generation of scripts, modification of ACCT into ACCT+S, source-level weaving, and the creation of instrumentation aspects for monitoring applications. Specification weaving of XTBL, a Web Service Management Language (WSML) document of performance policies and an XMOF document is currently partially implemented.

Near-term plans are to add to Mulini code to generate scripts that collect data from monitoring tools. This data will then be automatically placed in files and data analyzers generated. The analyzers will provide automatic assessment of whether the performance policies (and by extension the SLAs) have been met as well as how the system responds to changing workloads. Also, Mulini specification weaving will extend to allow multiple performance policy and deployment documents.

4.4.8. Evaluation

4.4.8.1. Scenario and Implementation

As an early experiment, is based on a well known application, the TPC-W benchmark, a transactional web e-commerce benchmark from the Transaction Processing Performance Council – TPC, as an exemplar for the automated staging approach. As mentioned, the tests use TPC-W as an illustrative mission-critical enterprise application, not for performance comparison of different platforms.

The TPC-W bookstore application was designed by the TPC to emulate the significant features present in e-commerce applications [41][72]. TPC-W is intended to evaluate simultaneously a vendor's packaged hardware and software solution – a complete, e-commerce system. Normally, benchmarks provide a preliminary estimate for vendors to compare system performance, but the TPC-W application also suggests measuring the resource utilization of subcomponents of the system under test, a concept which matches the goals in staging to uncover system behavior.

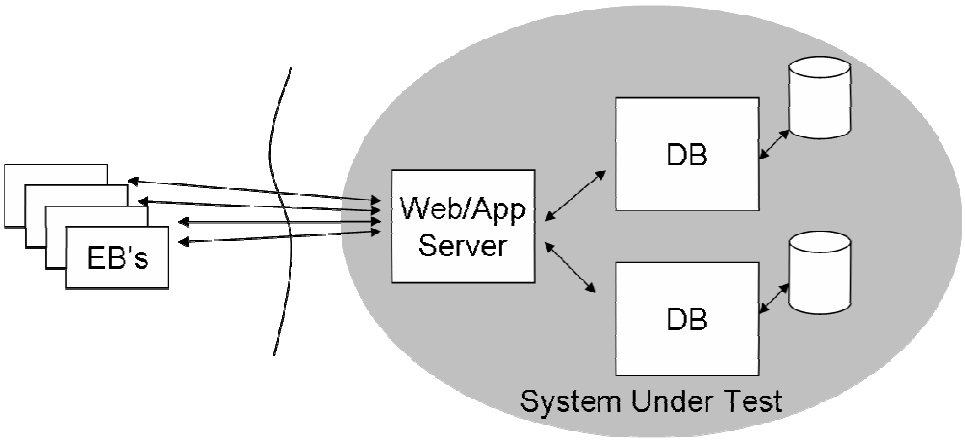


Figure 40. Simplified TPC-W application diagram. Emulated browsers (EB's) communicate via the network with the web server and application server tier. The application servers in turn are backed up by a database system.

The TPC-W application comprises two halves, as seen in Figure 40: the workload, which is generated by emulating users, and the system-under-test (SUT), which is the hardware and application software being tested. The system reuses and extends the software provided as part of the PHARM benchmark [82] and studies of performance bottlenecks and tuning in e-commerce systems [3][41]. The system tested employs a combined Java web/application server, using Apache Tomcat, and a database server, in this case MySQL.

In the TPC-W scenario, customers' navigation of the web pages of an online bookstore is simulated by remote emulated browsers (EB's). Each emulated browser begins at the bookstore's home page where it "thinks" for a random time interval after which the EB randomly chooses a link to follow. These link choices are from a transition table in which each entry represents the probability *p* of following a link or going offline. The specification provides three different models (that is, three different transition tables) each of which emulates a different prevailing customer behavior.

For the application, there are two primary metrics of concern: requests served per second, which is application throughput; and response time, which is the elapsed time from just before submitting a URL GET request to the system until after receiving the last byte of return data. The result of a TPC-W run is a measure called WIPS – Web Interactions per Second. There are several interaction types, each corresponding to a type of web page. For instance, a "Best Seller" interaction is any one of several links from the home page that leads to a best seller list such as best sellers overall, best selling biographies, or the best selling novels.

One test parameter is to select the number of concurrent clients (number of EB's). Varying the number of concurrent clients is the primary way of adjusting the number of web interactions per second submitted to the system. The most influential parameter is the number of items in the electronic bookstore offered for sale. This particular variation is also called the *scale* of the experiment and may be 1000, 10,000, or 100,000 items. Since many of the web pages are generated views of items from the database, normal browsing behavior can, excluding caching, slow the performance of the site. As an application parameter, scale level has the greatest impact on the performance of the system under test [41].

TPC-W v1.8 was chosen due to its widespread use in research and the availability of a reference implementation over the newer 2.0 benchmark which has yet to be fully ratified by TPC. The evaluations utilized only the "shopping" browsing model. This model represents the middle ground in terms of interaction mix when compared to the "order" and "browsing" models.

The implementation of the TPC-W bookstore is as Java servlets using the Apache project's Jakarta Tomcat 4.1 framework, communicating with a backend MySQL 4.0 database both running on Linux machines using a 2.4-series kernel. For all evaluations, the database, servlets, and images are hosted on local drives as opposed to an NFS or storage-server approach. As in other TPC-W studies, to speed performance additional indexes are defined on data fields that participate in multiple queries. Connection pooling allows reuse of database connections between the application server and database.

The testing employs two classes of hardware. Low-end machines, "L," are dual-processor P-III 800 MHz with 512 MB of memory, and they are assigned an approximate

value of $500 based on straight-line depreciation from a “new” price of $2000 four years ago. High-end, “H,” machines are dual 2.8GHz Xeon blade servers with hyperthreading enabled and 4GB of RAM, and they are assigned them an approximate value of $3500 each based on current replacement cost. Assigning cost values to each server is a convenient proxy for the cost of a deployment configuration. For instance, assigning a “2H/L” configuration of two high-end servers and one low-end server an approximate value of 2 × $3500 + $500, or $7500.

4.4.8.2. Automatic Staging for TPC-W

The first step in preparing TPC-W for automatic staging is to create the clients and select service-side tools for monitoring resource usage. Writing new clients allowed them to be encapsulated in XSLT templates to support generation by Mulini and therefore parameterization through TBL. Next created were the deployment documents and TBL specification that describe TPC-W staging.

Just as in ACCT, a MOF file contained the CIM description of the hardware and software needed to support the TPC-W application. Once given the MOF description, Cauldron used the MOF to map the software onto hardware and produce a workflow for deployment of the applications. This MOF file is translated into an XML document using a MOF-to-XML compiler resulting in XMOF.

Next, the SLAs were written for the TPC-W performance requirements. While future incarnations of Mulini will rely on documents *derived* from the SLAs, currently practice is to use the SLAs as a convenient specification format; WSML is a convenient format for encoding performance requirements pertaining both to customer performance data and to prescribe metrics for monitoring as performance policies pertaining only to

the system under test. There are 14 types of customer interaction in the TPC-W application. Each of these interactions has its own performance goal to be met which is detailed in Table 22 of Appendix A. For instance, to meet the SLA for search performance, 90% of search requests must complete the downloading of search results and associated images in 3 seconds.

Finally, the third specification document is the TBL specification of the staging process. TBL is directly convertible by hand to XTBL, an XML based format, suited as input for Mulini. TBL includes information that relates the staging test to performance guarantees and the specific deployment workflow. Example excerpts from each of the three specifications documents can be seen in Figure 50 of Appendix B.

4.4.8.3. Overhead Evaluation

The first evaluation is designed to show that there is reasonable overhead when executing staging tests that are generated. This is important because too high overhead could reduce the relevance of staging results. The baseline for overhead evaluation is the original reference implementation (non-Mulini generated). Since it is not possible to glean an accurate understanding of overhead when the system runs at capacity, the tests use much lower numbers of concurrent users. These two tests, shown in Figure 41 and Figure 42, are executed first on low-end hardware with 40 concurrent users and then on the high-end hardware with 100 concurrent users. `sar` and `top` gather performance data during the execution of the application of both the Mulini generated variant and the reference implementation. The tests show Mulini generated code imposes very little performance overhead in terms of resource usage or response time on application servers or database servers in both the L/L and H/H cases.

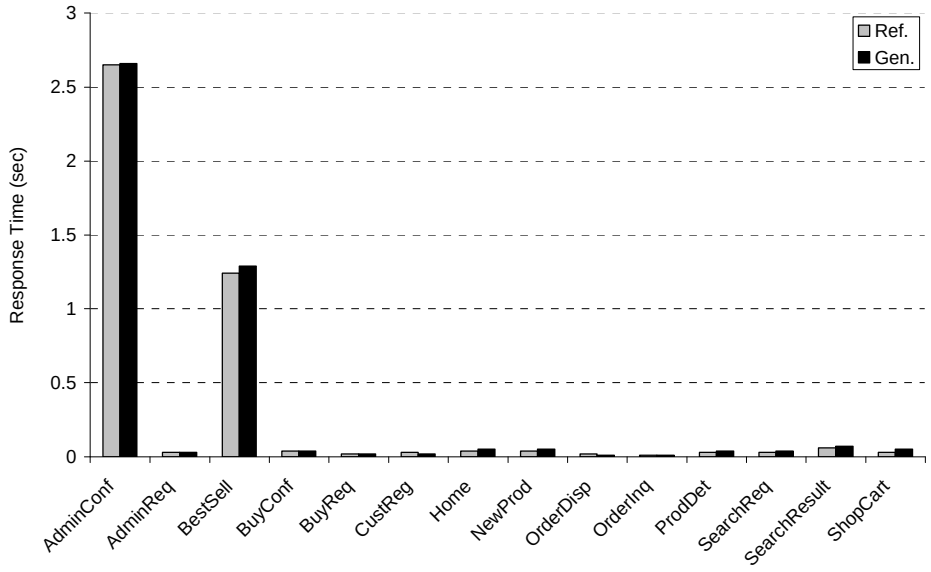


Figure 41. L/L reference comparison to gauge overhead imposed by Mulini with 40 concurrent users. In all interactions, the generated version imposes less than 5% overhead.

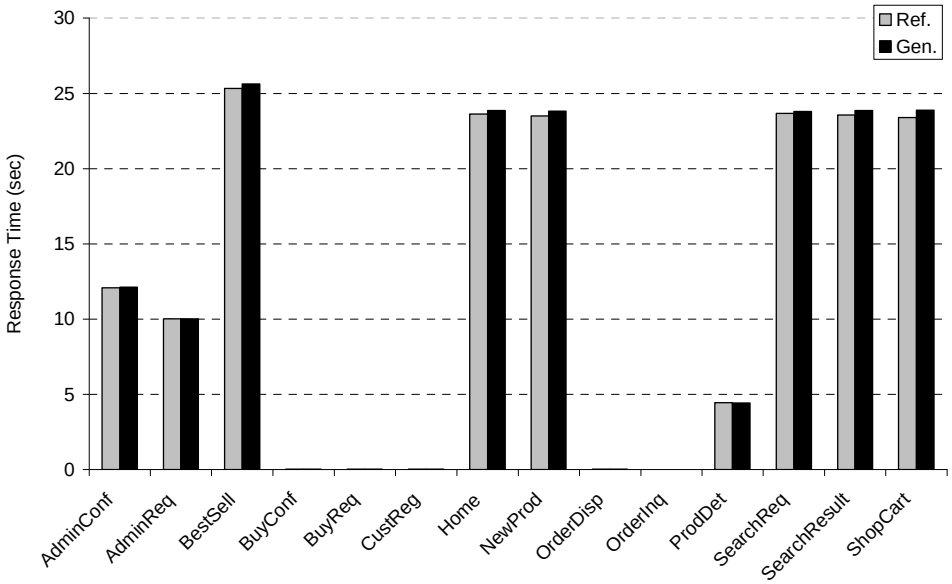


Figure 42. H/H reference comparison to gauge overhead imposed by Mulini with 100 concurrent users. For all interactions, the generated version imposes less than 5% overhead.

However, the target performance level for the TPC-W application is 150 concurrent users. This establishes that the generative techniques employed impose little overhead (<5%), leading next to measurements based on the application's formal design parameters.

4.4.8.4. Tuning TPC-W: Mulini in Use

In the next evaluations, Mulini-generated variants of TPC-W illustrate the utility of generated staging as compared to an "out-of-the-box" TPC-W. In fact, the performance monitoring tools of the primary experiment are wrapped for reuse in the generated scenarios. This way, they can become part of the automatic deployment of the TPC-W staging test. Being one of the more complex queries, it was shown in the initial test also to be longer running than most of the other queries.

Wrapping the performance tools used for monitoring performance is reasonably straightforward for a Clearwater-style generator. First, command-line scripts are constructed that execute the tools and then wrap these scripts in XSLT. This process that consists of adding file naming information and escaping special characters; these are added to XSLT templates to the main body of generator code. Once this is completed, it is easy to parameterize the templates by replacing text in the scripts with XSLT statements that retrieve the relevant information from the XTBL+ document.

This same technique performed to escalate database servlet code into the generator templates for direct instrumentation of their source. This is followed by adding an XML marker to denote a joinpoint in the code around the database query execution that should be monitored. An XSLT aspect template with XPath selects the marker and inserts timing code that implements measurement of the query.

Once aspect writing and template extension is complete, staging and tuning experiments executing the application can begin. Figure 43 shows the level of QoS satisfaction as specified by SLAs. Most of the high-end configurations perform well, while the low-end configurations have some problems. The raw data for this graph is available in Table 22 and

Table 23 in Appendix B.

The BestSeller transaction illustrates the differences among the configurations. Figure 44 zooms into Figure 43's third column from the left, showing very poor performance of the L/L configuration, very good performance of 2H/L and 2H/H (more than 90% satisfaction), with the H/L and H/H configurations in between (above 60% satisfaction) but still failing to meet the SLA.

To explain the differences in performance shown in Figure 44, it is necessary to examine the response time and throughput of the configurations via the direct instrumentation of the database servlet; woven aspect code measured the response time of a critical component of BestSeller interaction, the BestSeller database query.. The average response time is shown in Figure 45, where a clear bottleneck for the L/L configuration appears. Figure 45 shows that the response time of the BestSeller transaction is almost entirely due to the BestSeller Database Query, demonstrating the database to be the bottleneck. This finding is confirmed by Figure 46, which shows a marked increase in WIPS throughput when the database is moved to more powerful (and more expensive) hardware.

To migrate to more powerful hardware involved simply re-mapping the deployment to a high-end machine and re-deployed the staging and monitoring code automatically.

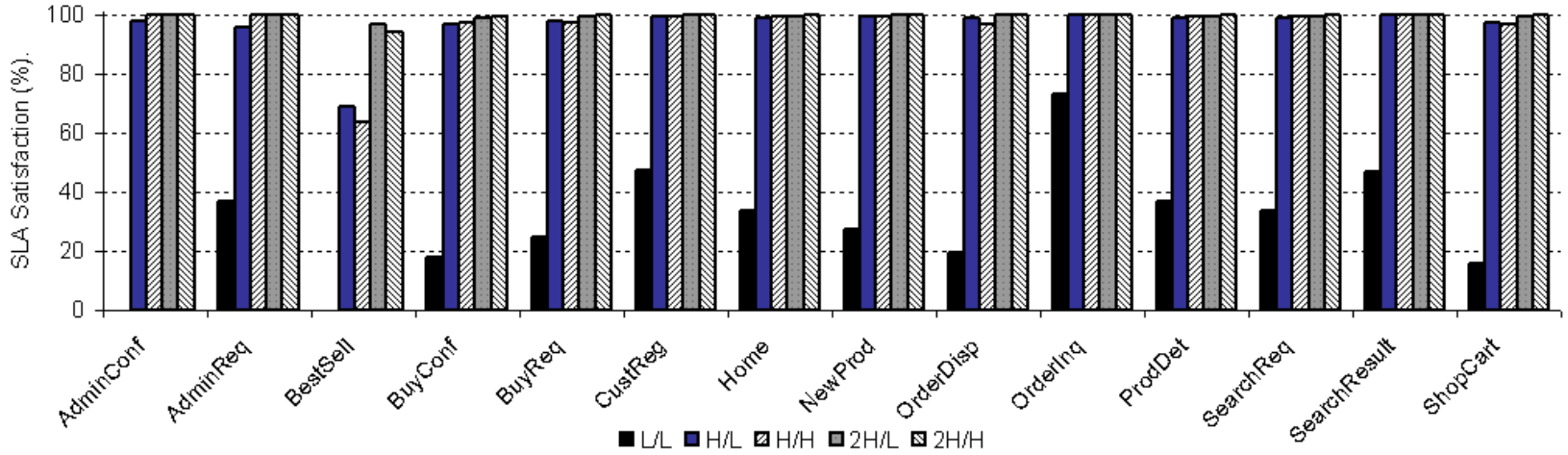


Figure 43. Summary of SLA satisfaction.

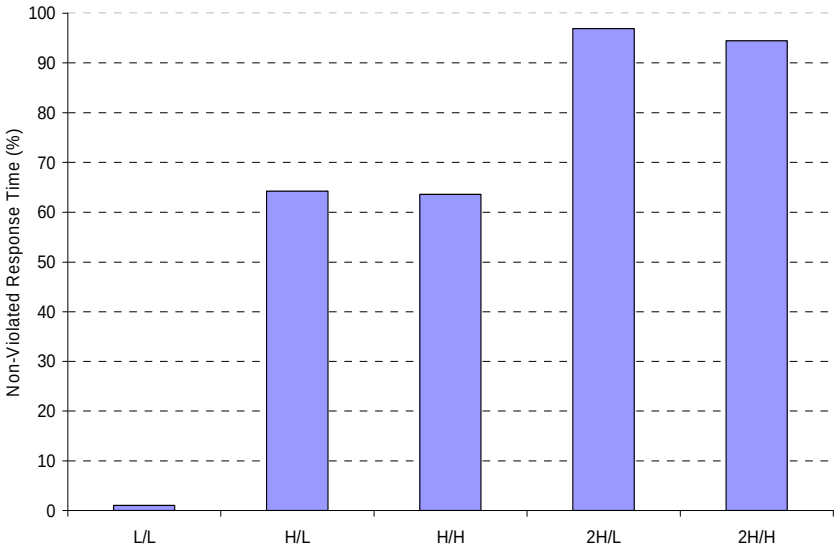


Figure 44. SLA Satisfaction of BestSeller

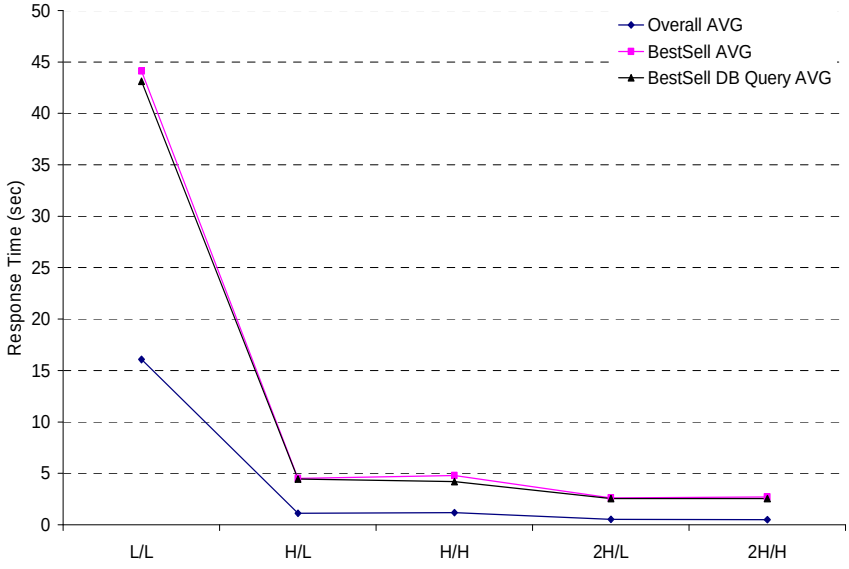


Figure 45. BestSeller average response time.

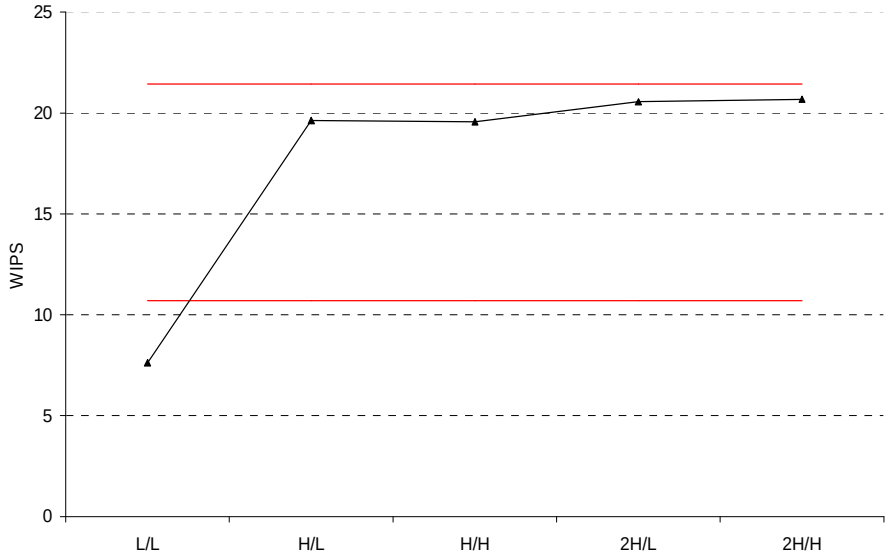


Figure 46. System throughput (WIPS). Lines above and below the curve demarcate bounds on the range of acceptable operating throughput based on ideal TPC-W performance as a function of the number of EB's.

The data shows that average response time of the query from the servlet to the database remains fairly long, indicating that even though the application server on low-end hardware is strained in terms of memory usage, the database remains the bottleneck even in cases of high-end hardware. Fortunately, MySQL allows database replication out-of-the-box. While this does not allow all database interactions to be distributed, it does allow "select" queries to be distributed between two machines, and these queries constitute the bulk of the TPC-W application. A straightforward re-write of the database connection code expands TPC-W to take into account multiple databases in the application servlet; this is followed by adding and modifying deployment to recognize the replicated database server. Using this method to allow the 2H/L and 2H/H cases, it was possible to create a system within the performance specification. To understand the

operating differences between deployments, it is instructive to examine the resource utilization reported by the monitoring tools.

First, while the database server uses only about 60% of the CPU, the *system* memory utilization consistently gets close to 100% due to filesystem caching as shown in Figure 47. This was ascertained by generating a script that measured actual process memory usage and comparing this data with the overall system memory usage reported by the kernel. As the daemon process for the servlets remained constant in size, it was apparent that application activity was exerting pressure through the operating system's management of memory. The memory and CPU utilization of the database server is plotted below in, showing the memory bottleneck in addition to the CPU bottleneck of the database server in the L/L configuration. (Note the included the approximate asset cost for each deployment.)

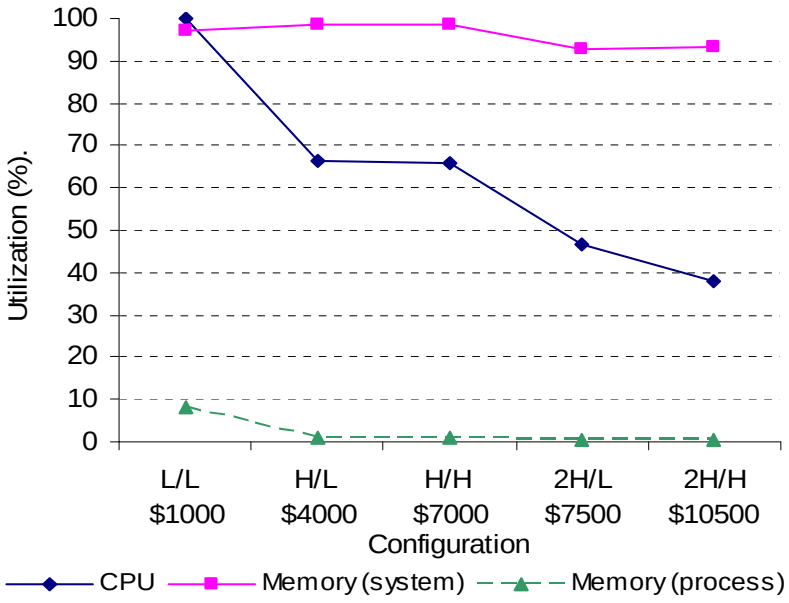


Figure 47. Database server resource utilization. The kernel's file system caching creates the spread between system memory utilization and the database process's memory utilization.

Resource usage for the application server is shown in Figure 48 again including approximate asset cost for the deployments. The figure shows consistent CPU and memory utilizations for high-end and low-end configurations. At around 20% CPU utilization and 15% memory utilization, the results indicate the low-end hardware is a viable application server choice for the target workload of 150 concurrent users, since the high-end configuration uses less than 5% of CPU resources with very little memory pressure (evidenced by system memory utilization being below 80%, a number which in such an experience is not atypical for a Linux system under only light load).

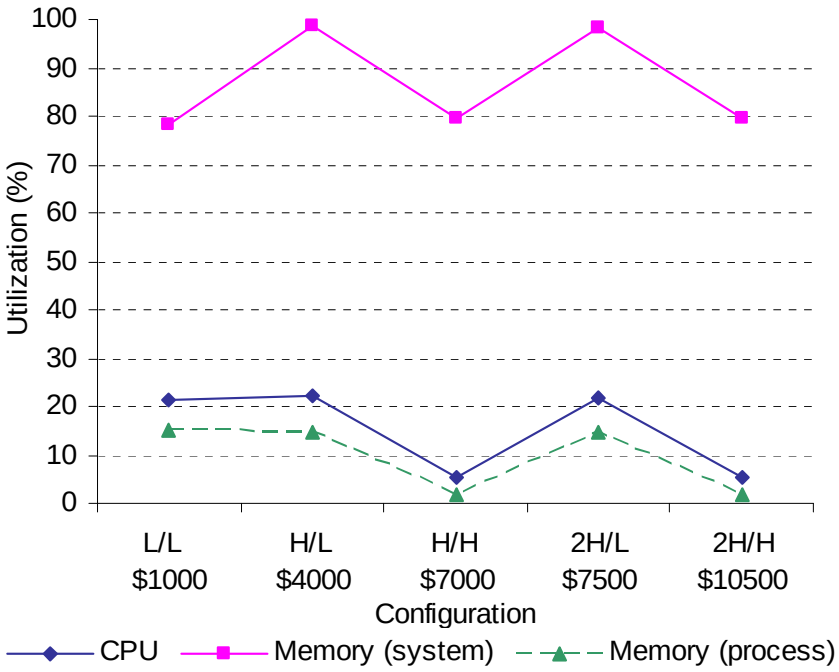


Figure 48. Application server resource utilization.

At this point, automated generation has enabled low-effort testing that begins to provide enough information on which system administrators may base deployment decisions. Referring back to the previous two figures, the TPC-W "application provider" now has a clearer picture of the cost of deploying his service; while technically two

configurations do meet the SLAs, there data suggests there choice for the final configuration. The administrator can either choose between a deployment at lower cost (2H/L) with less growth possibility, higher cost with ample resource overhead (2H/H), or request another round of staging (automatically) to find a better mix of the three machines that fulfill the SLAs.

4.5. Work Related to Distributed Enterprise Management

Recent years have seen the advent of wide-ranging resource management systems. For e-business, OGSA Grid Computing [40] aims to provide services within an on-demand data center infrastructure. IBM's Autonomic Computing Toolkit [52], the HP Utility Data Center [51] and Microsoft's DSI initiative [74] are examples of this. The distinction of this toolkit, however, is that Cauldron logic and a theorem prover to meet resource allocation constraints. There are several efforts related to specifying conditions and actions for policies, *e.g.*, CIM [34] and PARLAY [80]. However, it seems none of them have used a constraint satisfaction approach for automatic resource construction.

Another trend is deployment automation tools. Cfengine provides rich facilities for system administration and is specifically designed for testing and configuring software [16]. It defines a declarative language so that the transparency of a configuration program is optimal and management is separate from implementation. Nixes is another popular tool used to install, maintain, control, and monitor applications [35]. It is capable of enforcing reliable specification of component and support for multiple version of a component. However, since Nixes does not provide automated workflow mechanism, users manually configure the order of the deployments. For deployment of a large and complicated application, it becomes hard to use Nixes. By comparison, SmartFrog

provides control flow structure and event mechanism to support flexible construction of workflow.

Other projects address the monitoring of running applications. For instance, Dubusman, Schmid, and Kroeger instrument CIM-specified enterprise Java beans using the JMX framework [32]. Their instrumentation then provides feedback during the execution of the application for comparing run-time application performance to the SLA guarantees. In the Elba project, the primary concern is the process, staging, and follow-on data analysis that allows the application provider to confirm *before deployment* that the application will fulfill SLAs. Furthermore, this automated staging process allows the application provider to explore the performance space and resource usage of the application on available hardware.

Several other papers have examined the performance characteristics and attempted to characterize the bottlenecks of the TPC-W application. These studies generally focused on the effects of tuning various parameters [41], or on the bottleneck detection process itself [3][116]. The paper takes TPC-W, not as the benchmark, however, but as a representative application that allows illustrates the advantages of tuning applications through an automated process with feedback. The ActiveHarmony project also addressed the automated tuning of TPC-W as an example cluster-based web application [22]. While tuning is an important part of Elba, the Elba project stresses automation including design and deployment to the staging area by reusing top-level design documents.

Finally, there are also projects such as SoftArch/MTE and Argo/MTE that automatically benchmark various pieces of software [17][48]. The Elba emphasis,

however, is that this benchmarking information can be derived from deployment documents, specifically the SLAs, and then other deployment documents can be used to automate the test and staging process to reduce the overhead of staging applications.

4.6. ACCT and Mulini Summary

ACCT introduced the Clearwater approach applied to Automated Deployment of complex distributed applications. Concretely, ACCT translates Cauldron output (in XMOF format) into a SmartFrog specification that can be compiled into Java executables for automated deployment. ACCT performs a non-trivial translation, given the differences between the XMOF and SmartFrog models such as workflow dependencies. A demonstration application (JPetStore) illustrates the automated design and implementation process and translation steps, showing the increasing advantages of such automation as the complexity of the application grows.

The Elba project is a vision for automating the staging and testing process for enterprise applications. Elba efforts concentrate on mapping high-level staging descriptions to low-level staging experiments, dependency analysis to ensure proper component composition, and creating robust, adaptable designs for applications. Ultimately, long term efforts in Elba will be directed at closing the feedback loop from design to staging where knowledge from staging results can be utilized at the design level to create successively better designs.

The early results for Mulini reported here have shown promise in several areas. First, they show that Mulini's generative and language-based technique can successfully build on existing design (Cauldron) and deployment (ACCT) tools for staging. Second, experiences show that automatic deployment during the staging process is feasible, and

furthermore, that instrumentation of application code is feasible when using a Clearwater-based generator.

Ongoing research is addressing questions raised by these experiences and the limitations of the initial efforts. For example, one topic is exploring automated translation of SLAs into performance policies, which are translated into monitoring parameters to validate staging results. Another important question is the extension of TBL to support new applications. A related issue is the separation of application-dependent knowledge from application-independent abstractions in TBL and Mulini. A third question is the migration of staging tools and parameter settings (e.g., monitoring) to production use, so problems can be detected and adaptation actions triggered automatically during application execution.

CHAPTER 5
EVALUATING CLEARWATER: REUSE

5.1. Reuse Evaluations

Code generation is a code reuse strategy that when wedded to other advantages of domain specific languages, such as domain reasoning, verification, and simplification via abstraction, yields a powerful tool for programming. Therefore, one important evaluation for a code generation approach involves assessing the reuse within and across code generators. Unfortunately, unless there are very large sample sizes such metrics are difficult to assess, too, as they entail the measurement of actual use. Still, this chapter presents some early attempts to gauge reuse in Clearwater generators.

Particularly of interest in efforts to support research with code generators is the effort required to create a new output target. In such modification of a code generator, higher reuse within the generator implies more support for multi-platform development. To understand the importance of reuse in adding new targets, it is useful to walk through the abstraction process entailed in creating a code generator template.

Typically, the initial effort involves constructing a hand coded version implementing a specific application case. Then, this version must be re-implemented inside the generator while abstracting it a first time so that parameters in the code are mapped to values in the specification. The generator is then used to re-create the same code as the hand-written version and the new code can then be debugged. The result of this first iteration is a generator template that produces code of limited scope with respect to the domain.

Next, the code in the template is abstracted a second time into more general code that supports several generation cases. For example, in the first phase the generator developer may have only supported integers and their arrays, but in this second abstraction task, he adds support for floats, characters, and strings. Again the code must be tested and debugged. This means that there is generally high overhead in adding code to the generator since both the generator code and the generated code must be debugged.

This usage example brings up two different ways of looking at the complexity of Clearwater based generators. The first is on size alone – how big is the generator in terms of lines of code? Second, given the mode of evolution for generators presented above, reuse across platforms is the important feature. With this in mind, evaluation of reuse support in the ISG and ACCT/Mulini for creating new output targets can be accomplished by evaluating the sharing of code between different output platforms.

5.2. ISG

The ISG has been in development over a period of several years. Despite this, most efforts have concentrated on the continual refinement to develop the Clearwater code generation architecture. Even supporting several output options, the ISG is fairly manageable in terms of overall size, shown in Table 18 and Table 19 (calculated by David A. Wheeler's SLOCCount). Even though the Infopipes implementation is not complete, the generator still produces sophisticated code using a small codebase totaling fewer than 4500 total lines of C, C++, and XSLT. Furthermore, as illustrated in the examples and microbenchmarks, even this small codebase provides a good, basic, platform for information flow applications that can be customized with the application specific QoS. For instance, the simple image sending application's base code totaled

nearly 1000 lines and only relied on the C/sockets platform. Therefore, it is easy to see that the lines-of-code "investment" is worthwhile after creating even a few, simple applications.

Table 18. Lines of C++ code in language independent ISG modules excluding libraries (e.g., XSLT processor). This code is not in the templates and largely manages the generation process

Code (generation stage)	Line Count
Pre-process (1,2)	756
Generation (3, excl XSLT)	40
Weaver (4)	90
Write Files (5)	469
Shared all stages	134
Total	**1489**

Table 19. Lines of code (XSLT and target language) in XSLT templates that constitutes the language dependent modules of generation

Code (generation stage)		Line Count
master		56
C	core	276
	TCP	677
	ECho	475
C++	core	611
	TCP	638
C/C++ shared		211
Total XSLT		**2944**

One of the important features of having a pliable generator for Infopipes support is the capability to extend the generator piecewise to new targets efficiently. Such flexibility allows the ISG to support new language implementations which opens the ISG up to supporting future languages and legacy languages. Furthermore, supporting alternative middleware platforms allows Infopipes to adopt advanced middleware features, such as uploadable code for ECho, at a low cost.

Naturally, a source of well-debugged code is useful so that supporting new target platforms may be carried out efficiently as possible. With the Clearwater approach, extending to new platforms is greatly advantaged by the ability to share common code within each language. In Figure 49, gray bars represent the fraction of total lines that each communication layer and language platform as a portion of the total number of lines of XSLT within the code-producing portion of the generator.

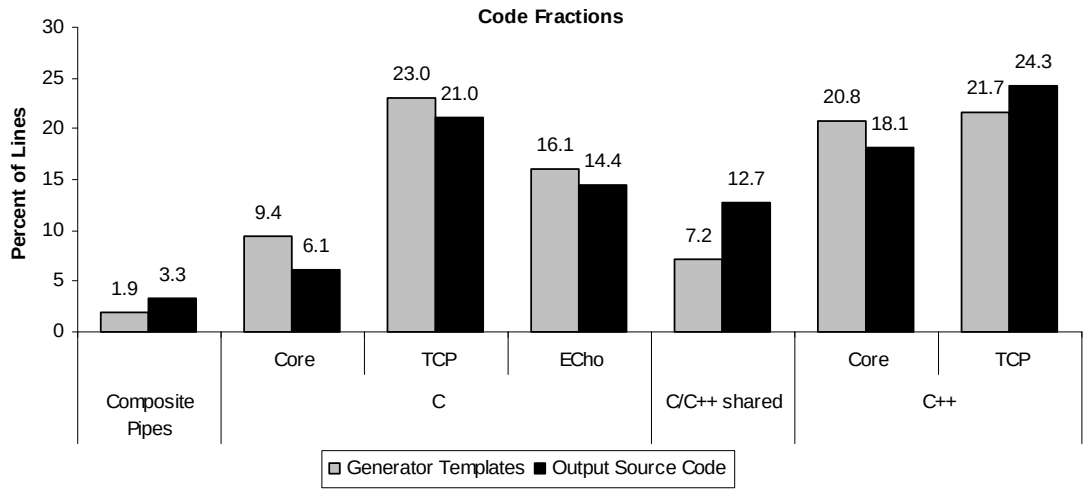


Figure 49. Fraction of code devoted to each platform mix of language and communication layer in both the generator templates ("Template") and the generated code ("Source").

It is first worthwhile to concentrate on the generator template fractions for C. The set of C templates comprises three different groups: those that correspond to "core" C services, such as a wrapper skeleton for the Infopipe middle code, and then two other groups of templates that support ISG communication code, TCP and ECho. Note that the fraction of lines dedicated to TCP and the number of template lines dedicated to ECho are not hugely different despite the fact that the two communication layers operate using vastly different semantics – bytes vs. events.

The second item of note is the fractions across language platforms, C and C++. First, the C++ core appears to use a great deal more effort that C. However, this is due to developer choice to include a class hierarchy for Infopipes as part of the generator. This reflects the Clearwater approach's support for idiomatic programming, *i.e.*, class-oriented modularity in C++ versus file-oriented in C. Furthermore, the TCP implementation on C++ closely matches that of the TCP implementation in C. This is due to the fact that much of the C code could be reused in the C++ implementation and modified to be encapsulated as part of a class while using class member accessor functions to read and write data to and from memory. Obviously, this reuse is useful in the fashion mentioned above – it need only be written and debugged once for both platforms. Additionally and importantly, this particular type of reuse enhances interoperability – the C and C++ code are guaranteed to have the same wire format.

Next, consider to the C/C++ shared category. This reflects code that is exactly the same in the C and C++ versions of the generator. This code implements utility functions for the Infopipes to write and retrieve runtime connection data from disk. Consequently, XSLT's included functionality allows both versions to be able to call the exact same template. (The C++ code can encapsulate the code in an "`extern`" block for proper name mangling.)

The final question relevant to code reuse in the ISG regards what fraction of code in the generator maps to what fraction of code in the output for an application. In other words, does the generator bias towards one platform or another? Measuring this involved devising a composition of Infopipes for each language/communication platform available. Then, SLOCCount provided a count of the lines of code in the unique files for

each platform. The unique files were then assigned to the categories corresponding to the generator categories. In Figure 49, these numbers are presented as the black bars. Note that these fractions generally correspond well across platforms. That is, regardless of platform, a given contribution to the generator will produce a similar contribution in the output code.

5.3. ACCT and Mulini

ACCT and Mulini are both significantly younger efforts than the ISG, and therefore it is more difficult to extract meaningful metrics. However, ACCT still provides some preliminary insight as to the code reuse within the generator.

Table 20. Code reuse within the ACCT generator.

Component Code	Target	Output Language	Lines of Code	Percent of Total
Java Harness	all	all	1463	64.7%
Templates	SmartFrog	Java	138	6.1%
		SmartFrog script	459	20.3%
	script	`bash`	200	8.8%
Total			**2260**	**100%**

Table 20 shows the size and fraction of the total generator for ACCT and its output templates. Clearly, the bulk of the code is contained in the reusable harness, which links all backends together. Especially interesting is that ACCT provides substantial new capability with the addition of the bash script backend without a large increase in the total lines of code.

Furthermore, as evidenced in the Mulini generator, the entire ACCT generator is reusable due to its support for extensible specifications and its pliability to extend with

new backends to support new output targets. In the creation of Mulini, this reduced the over 1600 lines of code in the harness and script backend to 200, an eightfold reduction. This code would otherwise have been written by hand from scratch. Even if an allowance is added for generator overhead of 50%, i.e. the generator itself demanded 800 of the 1600 lines of code, the reduction would still be four-fold, which is significant.

5.4. Code Reuse in Clearwater Summary

Even though the number of Clearwater-based generators is still small, such reuse numbers as are available suggest that these generators can reuse code efficiently. Particularly interesting is the reuse of ACCT within Mulini, wherein the extensible specification property is leveraged to incorporate the entire generator into a new tool.

Furthermore, reuse is likely to be enhanced through the availability of aspect oriented programming. Since customizations can be packaged separately from the generator, and since customizations can be built upon one another, the weaving capabilities of aspect weavers promise significant gains in code reuse for application developers, too.

Reuse evaluation will benefit as more applications, more researchers, and more code generators are constructed using Clearwater-approach techniques. In short, getting a truly clear picture of code reuse will take time.

CHAPTER 6
CONCLUSION

6.1. Contributions

The contribution of this dissertation is to show the practical utility of the Clearwater approach in solving problems encountered when applying domain specific languages to problems in distributed heterogeneous computing. By using XML, XPath, and XSLT, the Clearwater approach surmounts the challenges of abstraction mapping, heterogeneous interoperability, and flexible customization encountered in distributed computing problems. The contributions detailed in this dissertation are the Clearwater features allowing it to support extensible specifications, pliable generation, and flexible customization, and the application of these features in evaluation applications of two different distributed heterogenous systems domains. Finally, this dissertation takes a first look at reuse within the context of the Clearwater generation approach.

Distributed heterogeneous systems pose three challenges that appear inherently contradictory: the high abstraction and its mapping seem to oppose heterogeneous interoperability and flexible customization; similarly, flexible customization seems diametric to flexible customization. However, the Clearwater approach uses its three features (extensible specification, pliable generation, and flexible customization) to allow the creation of generators that can create customized generated code and yet evolve with the domain requirements. The challenges appear in the domains of information flow applications domain and the distributed enterprise application management, and this dissertation provides examples of the Clearwater approach used in these domains.

In the distributed information flow application domain, a horizontal domain in which the generator must create code for multiple applications, the ISG displays the ability of the Clearwater approach to generate code that matches the performance of several existing techniques for building communications into applications. Furthermore, two applications demonstrate the flexibility of the generator to generate code that supports application-specific QoS demands.

This is followed by the use of the Clearwater approach in a horizontal domain – generating applications to support distributed enterprise application management. Experiences with ACCT contribute examples of a flexible generator that can map to multiple, heterogeneous output languages. Mulini, which reuses ACCT, pushes further to support multiple input and output documents. Both generators have customization engines for creating customized code; furthermore, multiple types of output specifications may be customized (e.g., a shell script and Java source).

Finally, the last contribution of this thesis is a look at reuse within and across Clearwater generators. Interestingly, within the ISG generator, as the range of outputs was expanded, code could be reused from earlier platforms to ease development of the new output options. Mulini displayed reuse across generators by incorporating the ACCT generator within it as an entire module. Despite the small sample size, both of these cases point to high reuse efficiency for Clearwater based generators.

6.2. Open Issues

While Clearwater has been demonstrated with practical examples in two different problem domains, there remain several interesting open issues beyond the current

research into the spaces of aspect oriented programming, domain-specific language processing, and distributed heterogeneous systems.

6.2.1. Issues for AOP

The Clearwater-based research in this dissertation focused on the application of Aspect-Oriented techniques to output source-code and weaving new code into the existing, base output code. While a great deal of flexibility was recognized at this level of aspect weaving, as evidenced by the examples of customization of the example applications, the opportunity exists to apply the idea to the abstract specification level.

Extending the AOP capabilities to this layer of the code generation process should enable several advantages. First, constraints could be encoded as aspects that operate over an entire system specification. This work is similar to the work by Gray *et al* [46][47], but notably in the Clearwater architecture, it becomes possible to combine both the high-abstraction aspect-oriented customization and the source-code level customization across heterogeneous platforms using an XML based weaver for the source code level, and Clearwater supports extensible inputs. Such capability offers the possibility of performing the majority of the code generation with the weaving step. The chief advantage to this is that the entire body of code produced for an application target is tailored to that target – there is no extraneous code in the generation system. Such flexibility could lead to faster and more efficient generation and perhaps might lower the investment needed to extend the generator to new platforms or to include new capabilities. In fact, each new aspect could become a genuine extension of the generator.

6.2.2. Issues in Domain Specific Language Processing

The work presented in this dissertation concentrates solely on the code generation advantages offered by the Clearwater architecture. However, in the space of domain specific language processing there are several outstanding questions to be asked. For instance, one of the great strengths of domain specific language processing is that it enables domain specific analysis of a problem specification. That is, a candidate solution can be analyzed before being deployed or generated.

However, the feature of Clearwater generators to support extensible specifications makes such analysis more difficult. This difficulty arises because a generator may not be able to "see" the full specification if the specification has been extended beyond the capabilities of the domain analyzer. While resolution of this particular conflict may not be immediately apparent, there are candidate solutions. For instance, the extended specification may contain or reference additional code that allows the generator to analyze properly the extended sections. Alternatively, the generator may simply uses a guarding mechanism to check the specification contents against its own capabilities, and issue a warning or failure if the specification and generator version do not match.

In this case, too, we can see that the open issue of abstraction-level aspect-oriented programming may also provide a solution to the problem. Even if a generator were not to support all facets of the specification in question, a new aspect may be written that extends the generator to perform the evaluation.

6.2.3. Issues in Distributed Heterogeneous Systems

Finally, as is apparent in the experiments presented in the evaluation of the dissertation, the application of Clearwater to the systems arena is far from completely

explored. Information flow applications and the unique challenges they present are growing in importance as broadband network connectivity becomes more common. Furthermore, as enterprises, individuals, and devices create more information, flow-based processing of information is not just common, but important as well. Narrowing the scope to the enterprise space addressed by ACCT and Mulini, autonomous application management and policy-driven management tools continue to be important research and development areas as business manage ever increasing numbers of intereactions between computation elements. In all of these cases, heterogeneity continues to be a problem.

Within the scope of information flow activities, there are several Clearwater related issues to be explored. First and related to domain specific language processing, there are interesting questions raised by the current research as to how verified information flow systems can be assembled based on the current or foreseeable versions of the Clearwater approach. For instance, the current ISG performs no Infopipe-typespec checking. However, as pointed out in the discussion of system-level aspect oriented programming, such verification may be a suitable candidate for the additional functionality. Interestingly, using the aspect oriented solution to the problem promises not only a way to detect typespec mismatches but to inject abstraction mapping code that resolve or re-map inconsistencies.

Too, in information flow systems there is the question of more sophisticated management of information and information flows. For instance, the UAVDemo and Linear Road applications both showed the value of even simple flow management. More work research remains to determine the scalability of the approaches proposed within this work and the ability of the Clearwater architecture to accommodate them.

In the distributed enterprise application management space, there are significant opportunities to explore reuse as more applications and applications variations are incorporated for testing or deployment within the ACCT or Mulini generators. Web services are of particular interest within this domain.

Web services are grabbing the attention of commercial software developers and computer science researchers alike. The reasons for this interest stem from the same observations regarding problems in distributed heterogeneous computing that motivated the ISG, ACCT, Mulini, and the Clearwater approach: abstraction mapping, interoperable heterogeneity, and flexible customization. Thus, the results of Clearwater may be applicable to the web services space. In fact, one of the demonstration applications for the ISG relied on WSLA, a candidate service level agreement standard, to define quality of service constraints on an Infopipe.

In fact, the applicability of Clearwater to web service problems is supported by the image streaming application for Infopipes. The exact same framework used for implementing basic Infopipes and their functionality was used to implement support for part of the WSLA specification without modification. All that was required was to insert new XSLT templates that implemented the desired WSLA functionality as aspects. In Clearwater, furthermore, such functionality can be developed on an as-needed basis since a developer can choose which aspects to implement at which times and since XSLT templates can call other XSLT templates to form libraries of WSLA code generation functions.

Like the Mulini generator, web services have been adopting multiple specifications to capture features for web service applications. Naturally, leads to the

abstraction mapping problem which Clearwater addresses. For instance, web service developers recognize the need for constraints between parties involved in an application comprised from services such as service performance. This has led to the recognition and proposal of service level agreements (SLAs), which can capture expectations and roles in a declarative fashion [33][93]. One view of such additional specifications is that they constitute domain specific languages. As with all languages, then, the question becomes how to map the "high" abstraction of the service specification languages into a single lower-level implementation – the abstraction mapping challenge.

Web services have adopted XML document formats for information interchange. This is directly in response to the interoperable heterogeneity faced in distributed heterogeneous systems. By interposing XML between any two communicating web services, a great deal of platform variation, from operating system variations to language variations to location variation (firewall tunneling), can be concealed. On the other hand, it is clear that the community regards the problem as "unsolved" as there is research into methods to improve XML performance – such as compression to improve data transmission times [113].

In addition to viewing SLAs as a domain specific language, it is helpful to consider them as an aspect of a web-based application in the sense of Aspect Oriented Programming (AOP) [57]. This follows from noting that SLAs typically describe some application functionality that crosscuts application implementation which means that given a complete implementation of the application including service monitoring, then the SLA implementation code will be found in multiple components of the main

application, and furthermore, the crosscutting code is heavily mixed, or *tangled*, in components where this crosscutting occurs.

Given this view, these standards specify an aspect of their associated web services. That is, they capture a system characteristic that is orthogonal to the primary application functionality, and that functionality crosscuts, or touches on many parts of, the application's implementation. As a consequence, each of these standards can be seen as applying a customization to the underlying service implementations, just as QoS aspects in Infopipes provide customizations of the Infopipes communication base code. Therefore, web services face the heterogeneous customization challenge.

Given that these challenges are apparent for web services, performing research into applications of Clearwater code generation to the domain of web services and service oriented computing. Questions of interest include: Can AOP be supported at the domain specification level? Can specifications generally be implemented using AOP techniques? What types of specifications are best implemented via AOP versus explicitly adding such technology to the generator?

6.3. Finally

Clearwater has clearly shown itself to be useful and competitive for the domains examined thus far, and as noted, the Clearwater approach shows promise for Infopipes implementation and issues in distributed enterprise application management. Aside from the benefits from the generators themselves, already outlined, the implementations outlined above support some important conclusions about the approach itself.

From information flow applications, the ISG demonstrates some important properties. First, from a systems programming standpoint, Clearwater generator can

generate code that is performance competitive with code produced by existing code generation packages, i.e., the code generation technique is not going to trade generator flexibility and ease-of-programming for penalties on applications. Otherwise, the Clearwater approach might face resistance due to this performance penalty. Second, the generator is customizable across heterogeneous platforms a new advantage of existing code generation techniques. This places Clearwater as an option for a wide range of system types – a common situation as applications move from business-only or PC-only environments to run on platforms of divergent scale and paradigm, from Linux-on-mainframe to embedded Java on a cell phone.

For distributed enterprise application management, ACCT and Mulini have demonstrated the capacity for Clearwater based generators to address the abstraction mapping problem even when multiple abstract specifications are involved. Also, experiments with Mulini have shown that it is useful to incorporate new templates into the generator for application customization in a low-cost fashion, which Clearwater supports and was exhibited in experiments with instrumenting TPC-W code.

As with most research, Clearwater has raised questions are quickly as it provided answers, but already the work has shown that Clearwater approach answers affirms the thesis: using XML, XSLT, and XPath for code generation supports the building of code generators can meet challenges inherent in solving some problems found in distributed heterogeneous domains.

APPENDIX A
WOVEN CODE EXAMPLE

Below is sample code from the receiver side pipe function. To highlight the aspect code, ellipses stand in place of the application code with. It shows the additional include statements, timing code, and call to evaluate the SLA metrics:

```
__
control aspect (control_receiver.xsl)
timing aspect (timing.xsl)
cpu usage metric aspect (cpumon.xsl)
SLA evaluation aspect (sla_receiver.xsl)
__
#include "receiver.h"
#include "ppmIn.h"
#include "control.h"
#include <sys/time.h>
#include <stdio.h>
extern long usec_to_port_startup;
extern long usec_to_port_shutdown;
extern long usec_to_recv;
long usec_to_pipe_startup;
long usec_to_pipe_shutdown;
long usec_to_process;
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
float CPUUsage;
static long lastUTimeUse = 0;
static long lastSTimeUse = 0;
static struct rusage usingNow;
#include "sla.h"
int receiver( ) {
  ; // USER DECLARES VARS HERE
.
.
.
  struct timeval base;
  struct timeval end;
  gettimeofday(&base,NULL);
  ; // USER CODE GOES HERE
.
.
```

```
.
  gettimeofday(&end,NULL);
  usec_to_process = (end.tv_sec - base.tv_sec  ) * 1e6
                    + (end.tv_usec - base.tv_usec);
  fprintf(stdout,"Time to process: %ld\n",
          usec_to_process);
  getrusage( RUSAGE_SELF, &usingNow );
  CPUUsage = ((float) usingNow.ru_utime.tv_usec +
        usingNow.ru_stime.tv_usec - lastUTimeUse +
          ((float)  usingNow.ru_utime.tv_sec +
                    usingNow.ru_stime.tv_sec –
                              lastSTimeUse) * 1.0e6)
            / (usec_to_recv + usec_to_process);
  lastUTimeUse = usingNow.ru_utime.tv_usec +
                          usingNow.ru_stime.tv_usec;
  lastSTimeUse = usingNow.ru_utime.tv_sec +
                          usingNow.ru_stime.tv_sec;
  fprintf(stdout, "Use pct %0.2f.\n",
                                      CPUUsage * 100);
  processSLA();
  return 0;
}
```

APPENDIX B
RAW TPC-W EVALUATION DATA

Table 21. Resource utilization. "L" is a low end, "H" a high-end machine (see text for description). Percentages are for the system. "M/S" is "Master/Slave" replicated database

	DB host		APP server host	
	cpu(%)	**mem (%)**	**cpu(%)**	**mem (%)**
L/L	99.8	96.9	11.3	78.3
H/L	66.3	98.4	22.2	98.5
H/H	66.0	98.72	5.48	79.7
2H(M/S)/L	36.6/46.9	96.2/89.5	21.9	98.2
2H(M/S)/H	47.3/38.0	96.6/90.0	5.2	79.5

Table 22. Average response times. 90% WIRT is the web interaction response time within which 90% of all requests for that interaction type must be completed (including downloading of ancillary documents such as pictures). Each number is the average over three test runs. Even though some entries have an *average* response time that may be less than that in the SLA, the deployment may still not meet the SLAs 90% stipulation (e.g. "H/H" case for "Best Seller")

	Interaction													
Hardware Provision	**Admin Confirm**	**Admin Request**	**Best Seller**	**Buy Confirm**	**Buy Request**	**Customer Registration**	**Homepage**	**New Product**	**Order Display**	**Order Inquiry**	**Product Detail**	**Search Request**	**Search Result**	**Shopping Cart**
90% WIRT	20	3	5	5	3	3	3	5	3	3	3	3	10	3
L/L	54.6	9.3	44.1	18.9	11.1	6.9	9.0	12.6	11.6	3.10	8.04	8.6	12.6	14.6
H/L	7.2	0.4	4.5	0.7	0.3	0.2	0.3	0.4	0.2	0.04	0.2	0.3	0.42	0.39
H/H	7.8	0.2	4.8	0.7	0.4	0.2	0.2	0.4	0.3	0.01	0.2	0.2	0.39	0.4
2H/L	2.9	0.04	2.6	0.7	0.1	0.1	0.1	0.1	0.1	0.03	0.1	0.1	0.2	0.1
2H/H	2.8	0.04	2.7	0.4	0.1	0.1	0.1	0.1	0.1	0.02	0.1	0.1	0.1	0.1

Table 23. Percentage of requests that meet response time requirements. 90% must meet response time requirements to fulfill the SLA

L/L	0	36.8	0	17.5	24.6	47.1	33.7	27.3	19.3	73.4	36.9	33.7	46.6	15.5
H/L	97.8	95.7	68.8	97.1	97.7	99.5	99	99.3	99.1	100	99.2	99	99.9	97.2
H/H	100	100	63.6	97.2	97.2	99.6	99.4	99.4	96.9	100	99.3	99.3	99.9	97
2H/L	100	100	96.9	99.1	99.7	100	99.6	100	100	99.8	99.7	99.5	100	99.5
2H/H	100	100	94.5	99.7	99.8	100	100	100	100	100	100	99.9	100	100

APPENDIX C
XTBL, XMOF AND PERFORMANCE POLICY WEAVING

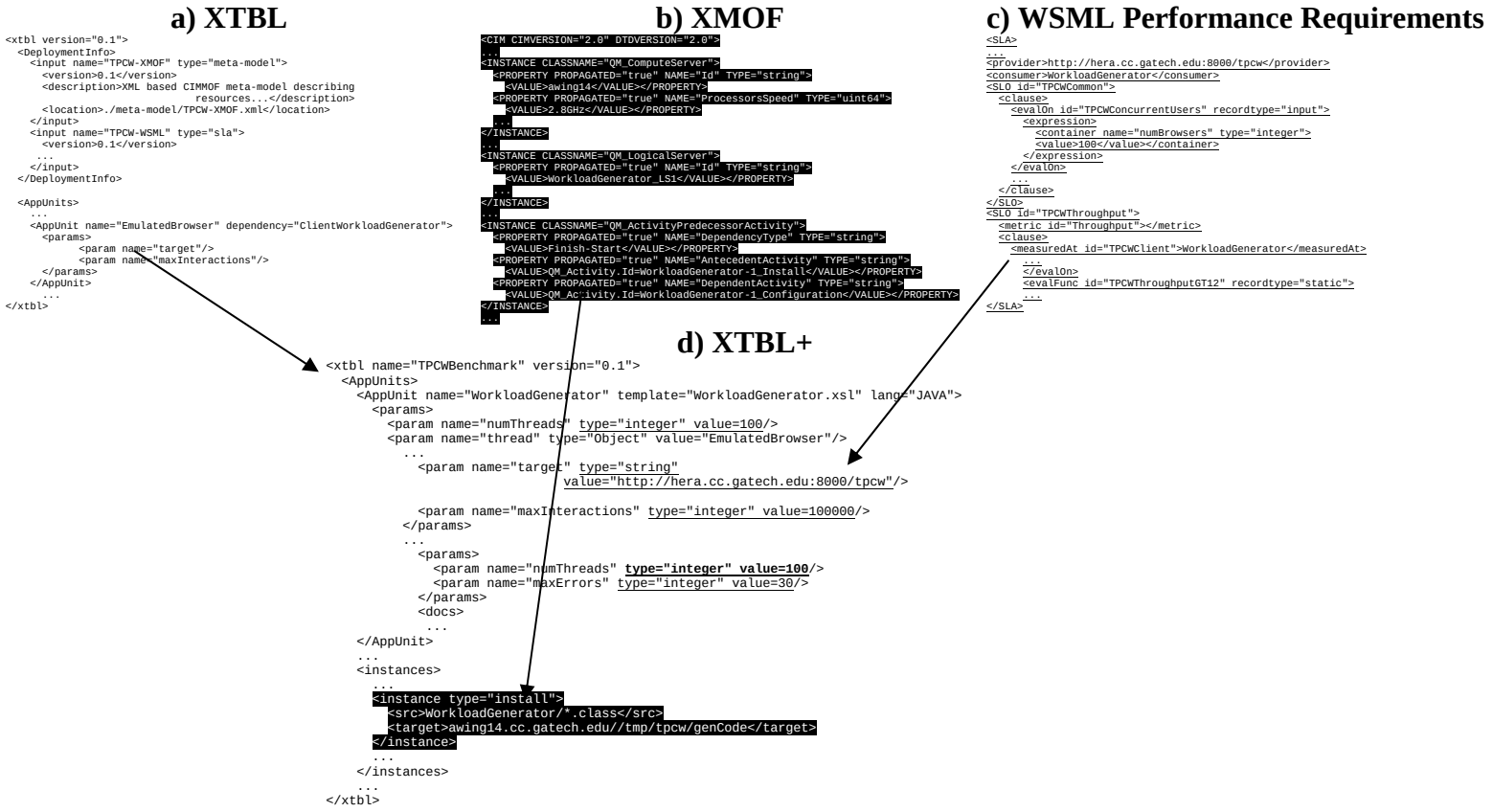


Figure 50. Example XTBL, XMOF, Performance requirements, and XTBL+. XTBL+ is computed from developer-provided specifications. XTBL contains references to deployment information, the XMOF (b), and to a WSML document (c), which encapsulates policy information applicable to staging. Mulini weaves the three documents into a single XTBL+ document (d).

REFERENCES

[1] Altova Mapforcehttp://origin.altova.com/products/mapforce/data_mapping.html. June 2006.

[2] Amagbégnon, P., Besnard, L., P. Le Guernic. Implementation of the data-flow synchronous language SIGNAL. PLDI 1995.

[3] Amza, C., Cecchet, E., Anupam Chanda, Sameh Elnikety, Alan Cox, Romer Gil, Julie Marguerite, Karthick Rajamani and Willy Zwaenepoel. “Bottleneck Characterization of Dynamic Web Site Benchmarks.” Rice University Technical Report TR02-388.

[4] Arasu, A., Babcock, B., Babu, S., J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, “STREAM: The Stanford Data Stream Management System”, *To appear in a book on data stream management edited by Garofalakis, Gehrke, and Rastogi*, 2004.

[5] Arasu, A., Cherniack, M., E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker and R. Tibbetts, “Linear Road: A Stream Data Management Benchmark”, Proceedings of the 30th International Conference on Very Large Data Bases (VLDB), August, 2004.

[6] Arasu, A., Babu, S., and J. Widom, “The CQL Continuous Query Language: Semantic Foundations and Query Execution”, Technical Report, Oct. 2003.

[7] Barreto, L., Douence, R., Muller, G., and Südholt, M. Programming OS schedulers with domain-specific languages and aspects: new approaches for OS kernel engineering. *International Workshop on Aspects, Components, and Patterns for Infrastructure Software* at AOSD, April 2002.

[8] BBN Technologies. QuO Toolkit Reference Guide. 2001.

[9] Benveniste, A., Caillaud, B., and P. Le Guernic. ``From synchrony to asynchrony." In J.C.M. Baeten and S. Mauw, editors, CONCUR'99, Concurrency Theory, 10th International Conference, vol. 1664 of Lecture Notes in Computer Science, 162-177. Springer V., 1999.

[10] Birrell, A., and Nelson, B. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems, 2, 1* (Feb. 1984), 39-59. Also appeared in Proceedings of SOSP’83.

[11] Black, P., Huang, J., Rainer Koster, Jonathan Walpole, Calton Pu, “Infopipes: an Abstraction for Multimedia Streaming”, in *ACM Multimedia Systems Journal*, 8(5): pp 406-419, 2002.

[12] Bonér, J., and Vasseur, A. AspectWerkz. http://aspectwerkz.codehaus.org/. June 2006.

[13] Bray, T., Hollander, D., and Layman, A. *Namespaces in XML*. http://www.w3.org/TR/REC-xml-names/. World Wide Web Consortium (W3C). 1999.

[14] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cohan, J., eds. *Extensible Markup Language*. http://www.w3.org/TR/xml11. World Wide Web Consortium (W3C). February 2004.

[15] Bryant, A., Catton, K., de Volder, G., and Murphy, C., "Explicit programming," *1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 22-26, 2002.

[16] Burgess, M., Cfengine: A Site Configuration Engine, USENIX Computing systems, Vol. 8, No. 3 1995.

[17] Cai, Y., Grundy, J., and Hosking, J. Experiences Integrating and Scaling a Performance Test Bed Generator with an Open Source CASE Tool. ASE 2004.

[18] Carney, D., Cetintemel, U., M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring Streams – A New Class of Data Management Applications", VLDB 2002.

[19] CDDLM Foundation Document. http://www.ggf.org/Meetings/ggf10/GGF10%20Documents/CDDLM_Foundation_Document_v12.pdf. June 2006.

[20] Chandrasekaran, S., Cooper, O., A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World". CIDR 2003.

[21] Chen, J., DeWitt, D., F. Tian, and Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases", SIGMOD 2000.

[22] Chung, I., and Hollingsworth, J. K. "Automated Cluster Based Web Service Performance Tuning." HPDC 2004. Honolulu, Hawaii.

[23] Clark, J. ed. *XSL Transformations*. http://www.w3.org/TR/xslt. World Wide Web Consortium (W3C). November 1999.

[24] Clark, J., and De Rose, S., eds. *XML Path Language* http://www.w3.org/TR/xpath. World Wide Web Consortium (W3C). November 1999.

[25] Coady, Y., Kiczales, G., M. Feeley, and G. Smolyn. "Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code," in

Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of Software Engineering, Vienna, Austria, 2001, pp. 88-98.

[26] Codesmith. http://www.ericjsmith.net/codesmith. June 2005.

[27] Common Object Request Broker Architecture (CORBA/IIOP). The Object Management Group. 1989-2002.

[28] Consel, C., Hamdi, H., L. Réveillère, L. Singaravelu, H. Yu, C. Pu. "Spidle: A DSL Approach to Specifying Streaming Applications". LABRI Research Report 1282-02.

[29] Consel, C., Hamdi, H., L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. "Spidle: A DSL Approach to Specifying Streaming Applications," in *Proceedings of the Second International Conference on Generative Programming and Component Engineering*. LNCS 2830, September 22-25, 2003, pp. 1-17.

[30] Czarnecki, K., and Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, June 6, 2000.

[31] Dan, A., Davis, D., D., R. Kearney, R., King, R., Keller, A., Kuebler, D., Ludwig, H., Polan, M., Spreitzer, M., and Youssef, A., Web Services on demand: WSLA-driven Automated Management, IBM Systems Journal, Special Issue on Utility Computing, Volume 43, Number 1, pages 136-158, IBM Corporation, March, 2004.

[32] Debusman, M., Schmid, M., and Kroeger, R. "Generic Performance Instrumentation of EJB Applications for Service-level Management." NOMS 2002.

[33] Debusmann, M., and Keller, A., "SLA-driven Management of Distributed Systems using the Common Information Model," *IFIP/IEEE International Symposium on Integrated Management*. March 2003.

[34] DMTF-CIM Policy. http://www.dmtf.org/standards/cim/cim_schema_v29. June 2006

[35] Dolstra, E., de Jonge, M., and Visser, E. "Nix: A Safe and Policy-free System for Software Deployment." In Proceeding of the Eighteenth Large Installation System Administration Conference. Atlanta, Georgia, 2004

[36] DVSL. http://jakarta.apache.org/velocity/dvsl/. June 2006.

[37] Eide, E., Frei, K., Ford, B., Lepreau, J., and Lindstrom, G. Flick: a flexible, optimizing IDL compiler. In *Proceedings of the 1997 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '97)*. Las Vegas, NV, Jun. 15-18, 1997.

[38] Eisenhauer, G. Bustamente, F., and Karsten Schwan. "Event Services for High Performance Computing," *Proceedings of High Performance Distributed Computing (HPDC-2000)*. August 2000.

[39] Eisenhauer, G., Bustamente, F., and Schwan, K. A middleware toolkit for client-initiated service specialization. *Proceedings of the PODC Middleware Symposium.* Portland, Oregon. July 18-20, 2000.

[40] Foster, I., Kesselman, C., Jeffrey M. Nick, and Steven Tuecke. "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration." Globus Project, 2002.

[41] García, D., and García, J. "TPC-W E-Commerce Benchmark Evaluation." *IEEE Computer*, February 2003.

[42] Gifford, D., and Glasser, N. "Remote Pipes and Procedures for Efficient Distributed Communication", in *ACM Transactions on Computer Systems*, Vol. 6, No. 3, August 1988, Pages 258-283.

[43] Global Grid Forum. http://www.ggf.org. June 2006.

[44] Gokhale, A.S., and D.C. Schmidt: Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems. *Proceedings of INFOCOM 1999*: 513-521. June 1999.

[45] Goldsack, P., Guijarro, J., Lain, A., Mecheneau, G., Murray, P., and Toft, P. SmartFrog: Configuration and automatic ignition of distributed applications. 2003 HP Openview University Association conference. 2003.

[46] Gray, J., J. Sztipanovits, D. Schmidt, T. Bapty, S. Neema, and A. Gokhale, "Two-level Aspect Weaving to Support Evolution of Model-Driven Synthesis." *Aspect-Oriented Software Development*. Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhan Clarke, eds. Addison-Wesley, 2004.

[47] Gray, J., T. Bapty, S. Neema, D.C. Schmidt, A. Gokhale, and B. Natarajan, "An Approach for Supporting Aspect-Oriented Domain Modeling," in *Proceedings of the Second International Conference on Generative Programming and Component Engineering*, LNCS 2830, September 22-25, 2003, pp. 151-168.

[48] Grundy, J., Cai, Y., and Liu, A. SoftArch/MTE: generating distributed system test-beds from high-level software architecture descriptions. In the *Proceedings of ASE 2001: The 16th IEEE Conference on Automated Software Engineering*. Coronado, CA. November 26-29, 2001.

[49] Hicks, M., Kakkar, P., Moore, J., Gunter, C., and Nettles, S. PLAN: A Programming Language for Active Networks. In the *Proceedings of the International Conference on Functional Programming (ICFP '98)*. ACM Press, September 1998, pp. 86-93.

[50] Hosoya, H., Vouillon, J., and Pierce, B. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46-90, 2005.

[51] HP Utility Data Center. http://www.hp.com/products1/promos/adaptive_enterprise/us/utility.html. June 2005

[52] IBM Autonomic Computing. http://www.ibm.com/autonomic. June 2006

[53] Jackson, A., Clarke, S. SourceWeave.NET: Cross-language aspect-oriented programming. In the *Proceedings of Generative Programming and Component Engineering: Third International Conference (GPCE 2004)*. Vancouver, Canada, October 24-28, 2004. Springer-Verlag LNCS 3286, 115 – 135.

[54] JBoss. http://labs.jboss.com/portal/jbossaop/index.html. June 2006.

[55] Karsai, G. Why XML is not suitable for semantic translation. Research note, Nashville, TN, April, 2000.

[56] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. "An Overview of AspectJ," in *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP), Budapest, Hungary, June 18-22, 2001, pp. 327-353.

[57] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In the *Proceedings of European Conference on Object-Oriented Programming, Aspect-Oriented Programming Workshop (ECOOP '97)*. Jyväskylä, Finland. June 10, 1997.

[58] Kiselyov, O., and Krishnamurthi, S. SXSLT: Manipulation Language for XML. In the Proceedings of *Practical Aspects of Declarative Languages: 5th International Symposium (PADL 2003)*. New Orleans, LA, January 13-14, 2003. Springer-Verlag LNCS 2562, 256-272.

[59] Koster, R., A. Black, J. Huang, J. Walpole and C. Pu, "Infopipes for Composing Distributed Information Flows". In the *Proceedings of the ACM Multimedia Workshop on Multimedia Middleware*, Ottawa, Canada, October 2001.

[60] Krishnamurthy, S., S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah, "TelegraphCQ: An Architectural Status Report". IEEE Data Engineering Bulletin, Vol 26(1), March 2003.

[61] Kulkarny, V., and Reddy, S. Separation of concerns in model-driven development. *IEEE Software*. 2003. Vol. 20, Issue 5. 64-69.

[62] Le Hégaret, P. *Document Object Model (DOM)*. http://www.w3.org/DOM/. World Wide Web Consortium (W3C). June 2006.

[63] Lieberherr, K. *Adaptive Object Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[64] Liu, L., C. Pu, R. Barga, and T. Zhou. "Differential Evaluation of Continual Queries". In IEEE Proceedings of the 16th International Conference on Distributed Computing Systems, Hong Kong, May 27-30 1996.

[65] Liu, L. C. Pu, and W. Han. "XWRAP: An XML-enabled Wrapper Construction System for Web Information Sources." In the *Proceedings of the 16th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, February 28 – March 3, 2000. pp. 611-621.

[66] Liu, L., C. Pu, and W. Tang, "Continual Queries for Internet Scale Event-Driven Information Delivery", *IEEE Transactions on Knowledge and Data Engineering*, Special issue on Web Technologies, Vol. 11, No. 4, July/August 1999.

[67] Liu, L., C. Pu, K. Schwan and J. Walpole: "InfoFilter: Supporting Quality of Service for Fresh Information Delivery", New Generation Computing Journal (Ohmsha, Ltd. and Springer-Verlag), Special issue on Advanced Multimedia Content Processing, Vol. 18, No. 4, August 2000.

[68] Liu, L., C. Pu, W. Tang, and W. Han, "Conquer: A Continual Query System for Update Monitoring in the WWW", Special issue on Web Semantics, *International Journal of Computer Systems, Science and Engineering*. 1999.

[69] Loyall, J.P., D.E. Bakken, R.E. Schantz, J.A. Zinky, D.A. Karr, R. Vanegas, and K.R. Anderson, "QoS Aspect Languages and Their Runtime Integration." *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98). Lecture Notes in Computer Science*, Vol. 1511, Springer-Verlag. Pittsburgh, Pennsylvania. May 28-30, 1998.

[70] Madden, S., M. Shah, J. M. Hellerstein, and Vijayshankar Raman, "Continuously Adaptive Continuous Queries over Streams", SIGMOD 2002.

[71] McNamee, D., J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet, "Specialization Tools and Techniques for Systematic Optimization of System Software", *ACM Transactions on Computer Systems*, Vol. 19, No. 2, May 2001, pp 217-251.

[72] Menascé, D. A. "TPC-W: A Benchmark for E-Commerce." *IEEE Internet Computing*. May-June 2002.

[73] Merillon, F., L. Reveillere, C. Consel, R. Marlet, G. Muller, "Devil: An IDL for Hardware Programming", in Proceedings of the 2000 Conference on Operating System Design and Implementation (OSDI), pp 17-30, San Diego, October 2000.

[74] Microsoft DSI. http://www.microsoft.com/management/. June 2006.

[75] Motwani, R., J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma,. “Query Processing, Resource Management, and Approximation in a Data Stream Management System”, In Proc. of the 2003 Conf. on Innovative Data Systems Research (CIDR), January 2003.

[76] Muller, G., R. Marlet, E.N. Volanschi, C. Consel, C. Pu and A. Goel, "Fast, Optimized Sun RPC using Automatic Program Specialization", *Proceedings of the 1998 International Conference on Distributed Computing Systems*, Amsterdam, May 1998.

[77] Nystrom, N., Clarkson, M. R., and Myers, A. C. Polyglot: an extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction* (Warsaw, Poland, April 2003). Springer-Verlag LNCS 2622, 138–152.

[78] Pal, P., J. Loyall, R. Schantz, J. Zinky, R. Shapiro, J. Megquier, " Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration." ISORC, March 2000.

[79] Papazoglou, M.. “Service-Oriented Computing: Concepts, Characteristics, and Directions.” Fourth International Conference on Web Information Systems Engineering (WISE'03). December 2003.

[80] PARLAY Policy Management. http://www.parlay.org/specs/. May 2005.

[81] Peterson, Larry, Tom Anderson, David Culler, and Timothy Roscoe. “A Blueprint for Introducing Disruptive Technology”, PlanetLab Tech Note, PDN-02-001, July 2002.

[82] PHARM benchmark. http://www.ece.wisc.edu/~pharm/tpcw.shtml. October 2005.

[83] Pu, C., and Swint, G. DSL weaving for distributed information flow systems (Invited Keynote). *Proceedings of the 2005 Asia Pacific Web Conference. (APWeb05)*. Shanghai, China. March 29 - April 1, 2005. Springer-Verlag LNCS. 2005.

[84] Pu, C., Galen Swint, C. Consel, Y. Koh, L. Liu, K. Moriyama, J. Walpole, W. Yan. Implementing Infopipes: The SIP/XIP Experiment, Technical Report GT-CC-02-31, College of Computing, Georgia Institute of Technology, May 2002.

[85] Pu, C., Galen Swint, Charles Consel, Younggyun Koh, Ling Liu, Koichi Moriyama, Jonathan Walpole, Wenchang Yan. "Implementing Infopipes: The SIP/XIP Experiment." Georgia Tech Research Report GIT-CC-02-31. Available :: http://www.cc.gatech.edu/projects/infosphere/papers/GIT-CC-02-31.ps

[86] Pu, C., K. Schwan, and J. Walpole, “Infosphere Project: System Support for Information Flow Applications”, in *ACM SIGMOD Record*, Volume 30, Number 1, pp 25-34, March 2001.

[87] Pu, C., T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole and K. Zhang, "Optimistic Incremental Specialization: Streamlining a Commercial Operating System", *Proceedings of the Fifteenth Symposium on Operating Systems Principles (SOSP’95)*, Colorado, December 1995.

[88] Pu, Calton, Galen Swint. "DSL Weaving for Distributed Information Flow Systems." (Invited Keynote.) *Proceedings of the 2005 Asia Pacific Web Conference. (APWeb05).* Springer-Verlag LNCS. March 29 - April 1, 2005. Shanghai, China.

[89] Rashid, A. Moreira, and J. Araújo. “Modularisation and Composition of Aspectual Requirements,” *1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 22-26, 2002.

[90] Sahai, A., Joshi, R., Singhal, S., and Machiraju, V. Automated policy based resource construction in utility computing environments. In the *Proceedings of the 2004 IEEE/IFIP Network Operations & Management Symposium (NOMS 2004).* (Seoul, Korea. April 19-24, 2004.)

[91] Sahai, A., Pu, C., Jung, G., Wu, Q., Yan, W., and Swint, G. Towards automated deployment of built-to-order systems. In *Proceedings of the 16th IFIP/IEEE Distributed Systems: Operations and Management (DSOM ‘05*) Barcelona, Spain. October 24-26, 2005.

[92] Sahai, A., Sharad Singhal, Rajeev Joshi, Vijay Machiraju. “Automated Generation of Resource Configurations through Policies.” IEEE Policy, 2004.

[93] Sahai, S. Graupner, V. Machiraju, and A. van Moorsel, “Specifying and Monitoring Guarantees in Commercial Grids through SLA,” *Third International Symposium on Cluster Computing and the Grid*. 2003.

[94] Salle, M., Sahai, A., Bartolini, C., and Singhal, S. A business-driven approach to closed-loop management. HP Labs Technical Report HPL-2004-205, November 2004.

[95] Sarkar, S., “Model Driven Programming Using XSLT: An Approach to Rapid Development of Domain-Specific Program Generators,” www.XML-JOURNAL.com. August 2002.

[96] Schonger, S., E. Pulermüller, and S. Sarstedt, “Aspect-Oriented Programming and Component Weaving: Using XML Representations of Abstract Syntax Trees,” *Proceedings of the 2nd German GI Workshop on Aspect-Oriented Software Development* (In: Technical Report No. IAI-TR-2002-1), University of Bonn, February 2002, pp. 59 – 64.

[97] Seri, M., T. Courtney, M. Cukier, and W.H. Sanders. An Overview of the AQuA Gateway. *Proceedings of the 1st Workshop on The ACE ORB (TAO),* St. Louis, MO, August 5-6, 2001.

[98] Shonle, M., K. Lieberherr, and A. Shah. XAspects: An Extensible System for Domain Specific Aspect Languages. OOPSLA 2003. October 2003.

[99] SmartFrog. http://www.hpl.hp.com/research/smartfrog/. June 2006.

[100] Steere, D., A. Baptista, D. McNamee, C. Pu, and J. Walpole, "Research Challenges in Environmental Observation and Forecasting Systems", *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom'00),* Boston, August 2000.

[101] SUN N1. http://www.sun.com/software/n1gridsystem/. June 2006.

[102] Swint, G., and Pu, C. Code generation for WSLAs using AXpect. *Proceedings of 2004 IEEE International Conference on Web Services (ICWS 2004)* (San Diego, California. July 6-9, 2004).

[103] Swint, G., C. Pu, and K. Moriyama, "Infopipes: Concepts and ISG Implementation," In Proceedings of the *2nd IEEE Workshop on Software Technologies for Embedded and Ubiquitous Computing Systems (WSTFEUS '04),* Vienna, Austria, May 11-12, 2004.

[104] Swint, G., Jung, G., and Pu, C. Event-based QoS for a distributed continual query system. *The 2005 IEEE International Conference on Information Reuse and Integration (IRI 2005).* Las Vegas, Nevada. August 14-17, 2005.

[105] Swint, G., Pu, C., Consel, C., Jung, G., Sahai, A., Yan, W., Koh, Y., Wu, Q. "Clearwater - Extensible, Flexible, Modular Code Generation." *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005).* Long Beach, California. November 7-11, 2005.

[106] Swint, G., Pu, C., Koh, Y., Liu, L., Yan, W., Consel, C., Moriyama, K., and Walpole, J. Infopipes: The ISL/ISG Implementation Evaluation. *Proceedings of the 3rd IEEE Network Computing and Application Symposium 2004 (IEEE NCA04).* Cambridge, Massachusetts. August 30 - September 2, 2004.

[107] Talwar, V., Dejan Milojicic, Qinyi Wu, Calton Pu, Wenchang Yan, and Gueyoung Jung. "Comparison of Approaches to Service Deployment." *ICDCS 2005*.

[108] Talwar, Vanish, Dejan Milojicic, Qinyi Wu, Calton Pu, Wenchang Yan, and Gueyoung Jung. "Comparison of Approaches to Service Deployment." *ICDCS 2005*.

[109] Thies, W., M. Karczmarek, and S. Amarasinghe. “StreamIt: A Language for Streaming Applications,” in *Proceedings of the 2002 International Conference on Compiler Construction*, LNCS, Grenoble, France. April 2002.

[110] Thies, W., M. Karczmarek, M. Gordon, D.Z. Maze, J. Wong, H. Hoffman, M. Brown, and S. Amarasinghe. “A Common Machine Language for Grid-Based Architectures,” *ACM SIGARCH Computer Architecture News*. New York. June, 2002, pp. 13-14.

[111] W3C Document Object Model. http://www.w3.org/DOM/. June 2006

[112] WBEM project. http://wbemservices.sourceforge.net.

[113] Werner, C., Buschmann, C. & Fischer, S. “WSDL-Driven SOAP Compression.” Pages: 18 – 35. University of Lubeck, Germany. JWSR Vol 2 No 1. Jan 2005.

[114] Wu, Q., C. Pu, A. Sahai. “DAG Synchronization Constraint Language for Business Processes.” In the *Proceedings of the IEEE Conference on E-Commerce Technology (CEC'06)*. June 26-29, 2006.

[115] Wu, Q., C. Pu, A. Sahai, R. Barga, G. Jung, J. Parekh, G. Swint. “DSCWeaver: Synchronization-Constraint Aspect Extension to Procedural Process Specification Languages.” In the *Proceedings of the IEEE International Conference on Web Services 2006 (ICWS 2006).* September 18-22, 2006.

[116] Zhang, Qi, Alma Riska, Erik Riedel, and Evgenia Smirni. “Bottlenecks and Their Performance Implications in E-commerce Systems.” *WCW 2004*. pp. 273-282. 2004.

[117] Zook, S. S. Huan, Y. Smaragdakis. “Generating AspectJ Programs with Meta-AspectJ.” *Proceedings of the Third International Conference on Generative Programming and Component Engineering (GPCE)*, Vancouver, Canada, October 24-28 2004.